
2. AUSGABE
REFERENZSEITEN FÜR
CBM · APPLE II · ATARI 400
COMMODORE 64 · ATARI 800
VC20 · APPLE II E · COMMODORE 600/700
6502-
ASSEMBLER-KURS FÜR BEGINNER
6502
ANDREAS DRIPKE
INTERFACE AGE

Dripke, Andreas
6502 ASSEMBLER-KURS FÜR BEGINNER

1. (1983)
2. (1983)
3. (1984)

ISBN 3-88623-018-x

Besonderer Dank gilt Herrn Michael Krause, ohne den die 'T.EX.AS.'-Konzeption einschließlich dieses Assembler-Kurses nicht zustandegekommen wäre.

Made in West Germany

Distribution (BRD): INTERFACE AGE VERLAG GMBH, Vohburgerstr.1
D-8000 München 21, Tel. (089) 5 80 67 02.

International Distribution: COMPUTER BOOKBASE
(DATA DYNAMICS TECHNOLOGY)
Div. of INTERFACE AGE Inc., USA
16704 Marquardt Avenue
Cerritos, CA 90701, U.S.A.

Gesamtherstellung: F. Hellmich, München

Vorwort

Assembler zu beherrschen bedeutet Zugriff zu haben zum „Herz“ des Computers. Assembler ist die einzige Programmiersprache, die den Mikroprozessor direkt ansprechen kann. Deswegen sind auch die besten Programme in Assembler geschrieben.

Jeder Mikroprozessor hat seine eigene Assemblersprache. Wir wollen uns in diesem Buch mit „6502-Assembler“ beschäftigen. Der 6502-Prozessor ist einer der am weitesten verbreiteten Mikroprozessoren – Sie finden ihn unter anderem in Computern von Commodore, Apple, Atari und vielen anderen Herstellern.

Vorkenntnisse in Assembler setzen wir bei Ihnen nicht voraus. Wir werden Ihnen – von Anfang an – den richtigen Weg zeigen. Grundkenntnisse in BASIC hingegen sind von Vorteil. Sie sind zwar nicht unbedingt zum Verständnis des Lehrstoffs erforderlich, wir werden aber doch hin und wieder Parallelen zwischen Assembler und BASIC ziehen, und da kommen Ihnen BASIC-Kenntnisse natürlich zugute.

Unser oberstes Ziel bei der Konzeption dieses Kurses war die leichte Verständlichkeit. Alle anderen Zielsetzungen – wie Systematik und Vollständigkeit – haben wir diesem Gesichtspunkt untergeordnet, ohne sie – wie wir glauben – zu vernachlässigen. Dabei werden wir in einer Mischung aus Theorie und Praxis nach und nach alle Möglichkeiten der Assemblersprache besprechen. Wir meinen, daß Sie auf diese Weise besonders schnell und effektiv den Lehrstoff beherrschen werden.

Die Darstellung der praktischen Beispiele bezieht sich durchweg auf das „'T.EX.AS.'-Assembler Entwicklungs- und Lehrsystem“, das ebenfalls bei uns erhältlich ist. Dieses System ist zwar nicht unabdingbare Voraussetzung zum Verständnis der Materie, es erleichtert Ihnen aber vieles. Darüberhinaus haben Sie mit 'T.EX.AS.' („Terminal Extended Assembler) ein äußerst vielseitiges Assembler-System, das Ihnen auch nach der Lernphase gute Dienste leisten wird.

Wenn Ihnen dieses Buch zusagt, freuen wir uns natürlich über Ihre Empfehlung an Freunde, Bekannte und Kollegen. Aber auch für Ihre Fragen und Kritik sind wir offen. Schreiben Sie uns: INTERFACE AGE VERLAG GmbH („COMPUTER LIFE“), Vohburgerstr. 1, D-8000 München 21. Und wenn Sie einmal ein wirklich besonders gutes Programm geschrieben haben, von dem Sie meinen, daß es auch für andere interessant sein könnte, informieren Sie uns; vielleicht nehmen wir es ja sogar in unseren Vertrieb auf.

Nun wollen wir Sie aber nicht länger mit Vorreden aufhalten sondern uns der Assemblersprache zuwenden. Viel Spaß dabei, und insbesondere – viel Erfolg!

Inhaltsverzeichnis

Um dieses Buch stets auf dem neuesten Stand halten zu können, trägt es keine Seitennumerierung, sondern eine Numerierung innerhalb der Kapitel. Das erlaubt uns, für neu erscheinende Computer Seiten einzufügen etc. Wir bitten Sie um Verständnis, letztendlich kommt Ihnen diese Aktualität zugute.

Kapitel 1: Warum Assembler?

Bevor wir uns gleich mit der Programmierung in Assembler im einzelnen auseinandersetzen, wollen wir erst einmal die Programmiersprache Assembler einordnen. Dann können wir nämlich auch die Frage beantworten, unter welchen Bedingungen die Programmierung in Assembler sinnvoll ist – und unter welchen nicht, denn auch das gibt es.

Man unterscheidet bei den Programmiersprachen grundsätzlich zwischen „maschinenorientierten" und „problemorientierten" („höheren") Sprachen. „Maschinenorientiert" meint, daß die betreffende Programmiersprache direkt vom Computer (von der Maschine) verstanden wird. Dazu ist auch Assembler zu rechnen Ein Assembler-Befehl kann direkt vom Mikroprozessor bearbeitet werden. Das bringt viele Vorteile – aber auch ein paar Nachteile. So erfordert die Programmierung in Assembler (oder wie man auch sagt: in „Maschinensprache) eine recht genaue Kenntnis der internen Struktur des verwendeten Mikroprozessors. Da aber jeder Prozessor eine jeweils eigene Struktur besitzt, muß der Programmierer oft eine ganze Reihe von – zum Teil sehr unterschiedlichen – Assemblersprachen beherrschen.
Wir werden hier „6502-Assembler" lernen. Wie Sie im Vorwort sicher schon gelesen haben, ist dieser Prozessor das „Herz" der Computer zum Beispiel von Commodore, Apple und Atari. Es gibt aber eine ganze Reihe anderer namhafter Hersteller, die andere Prozessoren verwenden und die wir nicht programmieren können jedenfalls nicht in 6502-Assembler. Bei aller Unterschiedlichkeit weisen diese verschiedenen Assemblersprachen jedoch auch eine ganze Reihe von gemeinsamen Strukturen auf, so daß – wenn Sie erst einmal eine der Assemblersprachen beherrschen – Sie die anderen wesentlich leichter lernen werden. Tatsache bleibt: Sie müssen für jeden Prozessor eine neue Sprache erlernen.

Damit kommen wir zu den „höheren" oder „problemorientierten" Programmiersprachen. BASIC zum Beispiel ist so eine. Oder auch PASCAL, FORTRAN, COBOL und andere. Ein BASIC-Befehl kann nicht direkt vom Mikroprozessor verarbeitet werden. Vielmehr muß die BASIC-Instruktion zunächst zerlegt und analysiert werden. Davon merken Sie natürlich im allgemeinen nichts, sondern Ihr Computer erledigt das selbständig. Man sagt auch der Computer „interpretiert" das BASIC-Programm und nennt den Teil innerhalb des Computers, der dafür verantwortlich ist. „Interpreter". Der Interpreter analysiert das BASIC Befehl für Befehl und setzt diese in interne Codierung um, die von dem Mikroprozessor verstanden werden können. Dieses Interpretieren kostet natürlich Zeit. Deswegen ist eine höhere Programmiersprache auch immer langsamer als eine maschinenorientierte. Der entscheidende Vorteil einer höheren Sprache ist aber, daß sie im allgemeinen auf einer Vielzahl von völlig unterschiedlichen Computern verfügbar ist. Programme in einer höheren Programmiersprache sind daher relativ einfach von einem zum anderen Computer übertragbar, bei maschinenorientierten Sprachen ist dies praktisch unmöglich – es sei denn, die beiden Computer haben zufälligerweise den gleichen Mikroprozessor.

Fassen wir die Unterschiede zwischen Maschinensprachen und höheren Sprachen zusammen:

Maschinenorientiert (Assembler):

Vorteile:

Die Befehle können direkt und ohne Umwandlung vom Mikroprozessor – dem „Herz“ jedes Mikrocomputers – verarbeitet werden.

Daher ist auch der Ablauf eines Assemblerprogramms sehr schnell.

Da alle Möglichkeiten des Prozessors optimal genutzt werden können, sind Assemblerprogramme – im allgemeinen – speicherplatzsparend und äußerst effizient.

Assemblerprogramme können auch auf sehr kleinen Computern ablaufen, da sie keinen komplizierten Interpreter o.ä. benötigen.

Nachteile:

Jeder Mikroprozessor hat eine andere Struktur und deswegen auch andere Befehlsworte.

Daher kann eine Programm, welches für einen bestimmten Prozessor geschrieben wurde, praktisch nicht auf einen anderen übertragen werden.

Die Assemblerprogrammierung ist oftmals mühselig, da der Programmierer sehr „maschinennah“ denken und programmieren muß.

Problemorientiert (höhere Sprache wie z.B. BASIC):

Vorteile:

Die Sprachen sind an eine menschliche Sprache angelehnt – meistens Englisch – und somit für einen Menschen leichter zu handhaben.

Die gängigen höheren Sprachen sind für praktisch jeden Computer verfügbar, Programme in diesen Sprachen können somit relativ leicht von einem Computer zum anderen übertragen werden.

Höhere Sprachen sind problemorientiert, d.h., es gibt verschiedene Sprachen z.B. für mathematische, für kaufmännische etc. Anwendungen.

Nachteile:

Höhere Sprachen benötigen einen Interpreter im Computer, der die betreffende Hochsprache in Maschinenanweisungen umsetzt.

Dadurch wird die Ausführungszeit eines Programms zum Beispiel in BASIC ganz erheblich verlangsamt.

Ein solcher Interpreter benötigt relativ viel Computerspeicherplatz.

Programme in einer höheren Sprache sind meist nicht so effizient wie Assemblerprogramme.

Sie sehen, es gibt eine ganze Reihe von Kriterien für die Wahl der geeigneten Programmiersprache. Um die Vorteile von höheren Sprachen und Assembler gleichzeitig ausnutzen zu können, bedient man sich oftmals sog. „Compiler" (sprich „Kompeiler"). Ein Compiler wandelt in einem Übersetzungsvorgang (genannt „Compilierung") ein zum Beispiel in BASIC geschriebenes Programm in Assembler um. Das hat den Vorteil, man kann in BASIC sehr komfortabel programmieren und erhält dennoch schließlich ein Assemblerprogramm. Ein solches durch Compilierung entstehende Assemblerprogramm wird aber niemals auch nur annähernd so effizient sein wie ein von Menschenhand geschriebenes.

Wir meinten, Ihnen diesen Überblick über die verschiedenen Ebenen von Computersprachen schuldig zu sein. Schließlich möchten wir nicht nur, daß Sie Assembler beherrschen, sondern die Sprache auch in größerem Rahmen einordnen können. Bleibt als Resumee festzuhalten, daß die Programmierung in Assembler nach wie vor dem, der sie beherrscht, viele Vorteile bringt. In Kürze werden auch Sie dazugehören.

Kapitel 2: Das erste Programm – LDA, STA, RIS

Nachdem Sie im ersten Kapitel einen ersten Eindruck von der Art der Assemblersprache bekommen haben, wollen wir jetzt gleich einmal ein kleines Beispielprogramm erstellen. Schließlich haben wir Ihnen eine Mischung aus Theorie und Praxis versprochen – kommen wir also jetzt zur Praxis. Bevor wir unser erstes Programm schreiben, müssen wir allerdings noch ein paar Grundsätzlichkeiten besprechen.

Wie Sie bereits wissen, ist Assembler eine maschinenorientierte Sprache, im Unterschied zum Beispiel zu BASIC. Nun sind aber die meisten heutigen Mikrocomputer – und insbesondere die schon etwas größeren – gar nicht mehr direkt für die Programmierung in Assembler gedacht. Paradoxerweise können diese Computer meist „von Natur aus" schon BASIC, wohingegen für die Programmierung in Assembler ein sog. „Assembler" notwendig ist. Machen Sie sich die Doppelbenutzung des Wortes „Assembler" klar. Zum einen verstehen wir unter Assembler die Programmiersprache als solches, zum anderen das Hilfsmittel, das es uns erlaubt, in Assembler (der Sprache) zu programmieren. Ein Assembler (Hilfsmittel) ist im allgemeinen ein Programm, das wiederum in Assembler (Sprache) geschrieben ist – so auch 'T.EX.AS.'.

Hier in diesem Buch sind alle Beispiele speziell auf das Assembler-System 'T.EX.AS' („Terminal Extended Assembler") abgestimmt. Sie können die Programme aber auch mit den meisten anderen verfügbaren Assembler-Systemen eingeben und austesten. Das System sollte allerdings einen sog. „Line-by-Line-Assembler" oder „Direkt-Assembler" beinhalten. Ziehen Sie am besten das jeweilige Handbuch mit zu Rate.

Noch ein Hinweis, wenn Sie mit einem VC-20 arbeiten: Die sog. „Bildschirmadressen" des VC-20 variieren in Abhängigkeit vom Ausbau der Speicherkapazität. Sehen Sie erforderlichenfalls in der VC-20-Referenz am Ende dieses Buches nach. Der letzte Satz gilt sinngemäß im Grunde für alle Computer. Alle hier im Buch beschriebenen Programme sind so ausgelegt, daß sie leicht in andere „Adressbereiche" gelegt werden können, wenn das erforderlich ist.

So, jetzt geht es aber los! Aktivieren Sie Ihr Assembler-System. Falls Sie schon ein wenig damit „gespielt" haben und noch Werte auf dem Bildschirm stehen, dann schalten Sie am besten Ihren Computer aus und ein und starten den Assembler nochmals, damit Sie auch exakt dieselben Ergebnisse erzielen wie wir hier.

Geben Sie sodann...

```
5120 LDA #65
```

..ein und drücken die RETURN-Taste, um diese Assembler-Anweisung zu übernehmen. 'T.EX.AS.' formatiert die Anweisung sofort und gibt sie in folgender Form wieder aus:

```
5120 LDA #65
```

Außerdem gibt 'T.EX.AS.' die nächste freie Adresse (was das genau ist, werden wir gleich besprechen) aus.

5122

Sollte das nicht passieren, so war Ihre Eingabe falsch. Korrigieren Sie in diesem Fall. Andere Assembler-Systeme als 'T.EX.AS.' haben unter Umständen gar keine automatische Formatierung oder Adressenvorgabe. Geben Sie in diesem Fall die Adresse 5122 von Hand ein.

Wir wollen nun unsere zweite Assemblerzeile abhängig vom verwendeten Computertyp eingeben:

Commodore CBM:	5122 STA 32768
VC-20 (bis 8K):	5122 STA 7680
VC-20 (größer 8K):	5122 STA 4096
Commodore 64:	5122 STA 1024
Apple II/IIe:	5122 STA 1024
Atari 400/800 (16K):	5122 STA 15424

(andere Atari-Modelle: DEEK (88) bringt die benötigte Zahl.)

(Hinweis: Bei der von Computer zu Computer variierenden Zahl handelt es sich um die erste Adresse des Bildschirm. Falls Sie mit einem völlig anderen Computer arbeiten als den hier genannten, schlagen Sie diese Zahl einfach in Ihrem Handbuch zum Computer nach. Wir werden hier in diesem Buch alle Programme beispielhaft für Commodore – dem im deutschsprachigen Raum am meisten verbreiteten – Computer auslegen und auch erklären. Für andere Systeme ergeben sich unter Umständen geringfügige Abweichungen.)

Vergessen Sie nicht, auch diesen Befehl durch Drücken der RETURN-Taste zu beenden. Bei 'T.EX.AS.' wird nun als nächste Adresse...

5125

..ausgegeben. Vervollständigen Sie diese Zeile mit...

5125 RTS

Damit haben wir unser erstes Programm schon fertig eingetippt. Sehen wir es uns noch einmal vollständig an:

```
5120 LDA #65
5122 STA 32768
5125 RTS
```

Die Zahl 32768 im zweiten Programmschritt ist speziell für einen Commodore CBM-Computer gedacht. Sie haben dort unter Umständen einen anderen Wert stehen (VC-20 zum Beispiel 7680 usw.).

Unser kleines Programm besteht aus drei Programmschritten. Jeder dieser drei Programmschritte definiert eine bestimmte Funktion in Assembler. Sie sehen bereits hier, daß Programmschritte unterschiedlich aufgebaut sein können. Allen dreien gemeinsam ist an erster Stelle eine sogenannte Adresse (5120, 5122, 5125) und an zweiter Stelle ein Assembler-Befehlswort aus je drei Buchstaben. Alle Befehlsworte in 6502-Assembler bestehen aus drei Buchstaben. Diesem Wort folgen im ersten und zweiten Programmschritt noch eine Zahl, im dritten nichts mehr. Diese dritte Zahl wollen wir „Argument“ nennen. Nun, es gibt ja auch in BASIC unterschiedlich lange Befehle.

5120	LDA	#65
Adresse	Assemblerbefehlswort (auch „Mnemonic“)	Argument

Doch bevor wir uns nun damit theoretisch auseinandersetzen, wollen wir erst einmal die Wirkung unseres kleinen Programms erkunden. Löschen Sie dazu den Bildschirm und bewegen den Cursor ein paar Zeilen nach unten. Bei Commodore 64 und VS-20 muß zunächst die Bildschirmfarbe verändert werden, damit überhaupt etwas sichtbar wird. Geben Sie dazu...

Commodore 64:	53281	7	(RETURN-Taste)
VC-20:	36879	152	(RETURN-Taste)

..ein. Tippen Sie nun – bei allen Serien – ...

EX 5120

.. ein und drücken wie üblich – RETURN. Unser Programm wird ab Adresse 5120 abgearbeitet. Wenn Sie alles richtig gemacht haben, müßte jetzt in der linken oberen Ecke des Bildschirms ein „A“ erscheinen. Ok? Sie können unser Programm so oft wie Sie wollen starten, die Wirkung ist stets die gleiche. „EX“ steht für „execute“, „ausführen“ und ist nur unter 'T.EX.AS.' verfügbar. Es wirkt ähnlich wie RUN bei BASIC-Programmen.

Von BASIC aus können Sie unser Programm jederzeit mit...

SYS 5120 ...bzw.... CALL 5120

.. aufrufen (SYS: Commodore, CALL. Apple; Atari: USR-Befehl).

Beachten Sie bereits an dieser Stelle unbedingt den Unterschied zwischen den Assembler-Befehlsworten und 'T.EX.AS.'-Kommandos. Ein Assembler-Befehlswort besteht wie Sie bereits wissen immer aus drei Buchstaben (zum Beispiel LDA, STA oder auch RTS). Diese Befehlsworte sind international standardisiert für den 6502-Prozessor. 'T.EX.AS.' kennt aber zusätzlich eine ganze Reihe von Kommandos, die der Handhabung oder dem Testen von Assemblerprogrammen dienen. Ein solches Kommando ist zum Beispiel „EX“ zum Ausführen von Programmen.

Um die erste Zahl eines jeden Programmschritts – eben die „Adresse" – verstehen zu können, müssen wir ein bißchen weiter ausholen. Wie Sie vielleicht wissen, besteht der Computerspeicher aus lauter einzelnen Zellen. Eine solche Zelle ist gerade so groß, daß sie ein einzelnes Zeichen speichern kann. Jeder Computer hat eine Vielzahl von Zellen, die zum Teil verschiedene Aufgaben erfüllen.

So ist zum Beispiel in einem Teil der Zellen das sog. „Betriebssystem" abgespeichert. Dieses Betriebssystem sorgt dafür, daß der Computer versteht, wenn wir auf der Tastatur etwas eingeben und dies entsprechend auf dem Bildschirm darstellt. Bei den meisten Geräten beinhaltet das Betriebssystem auch noch einen Interpreter, zumeist für BASIC. Sie errinnern sich noch, was wir unter „Interpreter" verstehen? Andere Zellen speichern unser Assemblerprogramm ab, wieder andere stellen den Bildschirm dar usw.

Um nun diese Vielzahl von Speicherzellen gezielt ansprechen zu können, hat man ihnen Zahlen zugeordnet. Man beginnt bei null zu zählen bis hin zur maximalen Anzahl von Zellen. Bei den meisten 6502-Mikrocomputern existieren insgesamt genau 65536 verschiedene Speicherzellen, die Numerierung läuft also von 0 bis 65535. Und da eine solche Nummer oder Zahl angibt, wie und wo eine bestimmte Speicherzelle angesprochen werden kann, hat sich dafür der Ausdruck „Adresse" eingebürgert. Eigentlich ganz einfach, nicht wahr?

Die erste Zahl eines jeden Programmschritts sagt uns also, wo, d.h. an welcher Adresse im Speicher, der Befehl steht. Unser Programm steht somit im Adressbereich von 5120 bis 5125. Sie werden erkennen, daß dies eine gewisse Ähnlichkeit mit der Verwendung von Zeilennummern in BASIC hat.

Nachdem wir nun wissen, was es mit der jeweils ersten Zahl auf sich hat, wenden wir uns nun den eigentlichen Assemblerbefehlen zu. Zum Beispiel für...

```
CBM:  LDA   #65          Commodore 64,   LDA   # 65
      STA  32768         Apple II/IIe:   STA  1024
      RTS                                RTS
```

Das Befehlswort „LDA" steht als Abkürzung für „load accumulator", übersetzt „lade Akkumulator". Und zwar wird der Akkumulator mit dem Wert 65 geladen. Was ein Akkumulator eigentlich ist?
Nun, von der BASIC-Programmierung sind Sie sicherlich den Umgang mit Variablen gewöhnt. Variablen sind Namen, die beliebige veränderliche – eben variable – Werte annehmen können. Der Wert kann jederzeit mit dem entsprechenden Variablennamen abgerufen werden. In Assembler gibt es solche Variablen nicht. Als einen – allerdings recht dürftigen – Ersatz existieren in Assembler drei sog. „Register". Diese drei Register tragen die Bezeichnungen „Akkumulator", „X-Register" und „Y-Register". Sie stellen eine Art Arbeitsvariablen dar. Der Ausdruck „Akkumulator" ist aus dem Wort „to accumulate" (ansammeln) entstanden.

Die Anweisung „LDA #65“ bedeutet „Lade den Akkumulator mit dem Wert 65“. Sie erinnern sich, die 65 nennen wir hierbei „Argument“. Das sog. Doppelkreuz davor bedeutet, daß der Akkumulator direkt mit dem Wert 65 geladen wird, und nicht zum Beispiel mit dem Wert aus Speicherzelle 65, was auch möglich wäre.

Alle drei der genannten Register können aufgrund ihrer Struktur nur Werte von 0 bis 255 annehmen, negative Zahlen oder solche größer als 255 sind also nicht erlaubt. Richtig sind somit Anweisungen wie...

```
LDA  #65    ;Anweisung aus unserem Programm
LDA   #0    ;lädt Akkumulator mit dem Wert 0
LDA #255    ;lädt Akkumulator mit dem Wert 255
```

.. wohingegen diese Befehle nicht korrekt sind und vom Mikroprozessor auch nicht verstanden werden:

```
LDA  #-1    ;negative Werte sind nicht erlaubt
LDA #300    ;Argument ist zu groß
LDA §256    ;dito, die Zahl ist größer als 255
```

Sie können sich bereits an dieser Stelle merken, daß der 6502 Mikroprozessor grundsätzlich nur mit Zahlen von 0 bis 255 arbeiten kann. Lediglich der Adressbereich reicht darüber hinaus von 0 bis 65535, wie wir bereits wissen. Jedoch wird auch das nur durch einen Trick erreicht, den wir später noch kennenlernen werden.

Um Ihnen den Einstieg in Assembler zu erleichtern, wollen wir unser Programm Schritt für Schritt mit entsprechenden Befehlen der Programmiersprache BASIC vergleichen. Für den ersten Befehl können wir schreiben:

```
LDA #65  .. entspricht etwa ...   A=65
```

Noch deutlicher wird die Bedeutung von „load accumulator“, wenn wir es wie folgt schreiben:

```
LDA #65  .. entspricht etwa ... LET A=65
```

Die Anweisung LET ist in BASIC zwar ungebräuchlich, aber dennoch erlaubt.

Der zweite Befehl in unserem Assemblerprogramm lautet ...

```
5122 STA 32768
```

(32768 für CBM). Die Adresse 5122 brauchen wir nicht mehr zu besprechen, um ihre Bedeutung wissen wir bereits.

Das Befehlswort „STA“ bedeutet nun „store accumulator“, „speichere Akkumulator ab“. Der Akkumulatorinhalt, in unserem Beispiel die 65, wird in die nach STA angegebene Speicherzelle abgespeichert. Es wird also die Zahl 65 in Speicherzelle 32768 bezw. 7680 transportiert. Beachten Sie, daß der Akkumulator selbst aber auch weiterhin noch die Zahl 65 beinhaltet. Es wird also im Grunde nur eine Kopie aus dem Akkumulator in die nach STA angegebene Speicherzelle gegeben.

Da das Argument zu STA, die 32768 bzw. 7680, eine Speicherzelle bezeichnet, nennen wir es – wie bei allen Speicherzellen – Adresse. Sie sehen, Adressen dienen nicht nur als Quasi-Zeilennummern für Assemblerprogramme, sondern es können auch Daten (Zahlen) dort abgespeichert werden. Das ist eigentlich auch einsichtig, wenn Sie sich verdeutlichen, daß ja der gesamte Computerspeicher ausschließlich aus einzelnen Zellen besteht. Diese werden daher sehr vielseitig verwendet.

Die dem STA-Befehl in BASIC entsprechende Anweisung ist der POKE-Befehl. Mit POKE können wir Werte in bestimmte Speicherzellen schreiben. Für unseren Vergleich mit BASIC gilt also...

STA 32768 .. entspricht (CBM) .. POKE 32768,A

.. bezw. ...

STA 1024 .. entspricht (C=64, Apple) .. POKE 1024,A

Der Inhalt des Akkumulators bezw. der Variablen A wird der Speicherzelle 32768 bezw. 1024 übergeben.

Kommen wir nun zu dem dritten und letzten Programmschritt unseres Assemblerprogrammes:

5125 RTS

„RTS“ steht als Abkürzung für „return from subroutine“, etwa „Rückkehr aus Unterroutine“. Wir wollen RTS an dieser Stelle als dem BASIC-Befehl END gleichwertig ansehen. RTS bewirkt also den Abschluß eines Assembleprogramms und Rückkehr nach 'T.EX.AS.' oder BASIC; je nachdem, von wo aus das Programm aufgerufen wurde („EX“ oder „SYS“ bezw. „CALL“).

RTS .. entspricht (hier!) ... END

Da keine Daten in irgendeiner Form verändert werden, benötigt RTS auch kein Argument, im Unterschied zu LDA und STA. RTS kann allerdings in Verbindung mit Unterprogrammen in Assembler noch weitreichende Funktionen erfüllen, wir wollen es aber zunächst bei dem Gesagten belassen. Schließlich können Sie nicht alles auf einmal lernen.

Immerhin sollten Sie sich schon an dieser Stelle merken, daß ein Assemblerprogramm in jedem Fall mit dem Befehl RTS abgeschlossen werden muß. Dies stellt einen Unterschied zu BASIC dar, wo die END-Anweisung ja entfallen kann.

So, damit haben wir unser erstes Assemblerprogramm erstellt, ausprobiert und ausführlich besprochen. Sie sehen, soooo schwer ist Assembler gar nicht.

Zusammenfassung

Unter dieser Überschrift werden wir von jetzt an in jedem Kapitel die wichtigsten Lerninhalte noch einmal in prägnanten Sätzen zusammenfassen.

Jeder Computerspeicher besteht aus einzelnen Speicherzellen, die von 0 bis 65535 durchnumeriert sind. Diese Nummern werden Adressen genannt.

Ein Assemblerprogrammschritt besteht aus einer Adresse, einem Befehlswort aus drei Buchstaben und ggf. einem Argument. Das Argument kann ein direkter Wert (Zeichen „§") oder aber eine Adresse sein.

Wir haben die folgenden Assemblerbefehle kennengelernt:

```
LDA §argument
STA adresse
RTS
```

LDA lädt den Akkumulator mit dem als Argument vorgegebenen Wert, der zwischen 0 und 255 – jeweils einschließlich – liegen darf. STA speichert diesen Wert im Akkumulator an der angegebenen Adresse ab, im Akkumulator selbst bleibt der Wert dabei erhalten. RTS beendet das Assemblerprogramm und bewirkt die Rückkehr nach BASIC oder 'T.EX.AS.'.

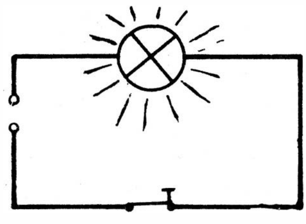
Von Basic aus wird ein Assemblerprogramm mittels „SYS startadresse" bezw. „CALL startadresse" gestartet, von 'T.EX.AS.' aus mit „EX startadresse".

Kapitel 3: Von Bits und Bytes

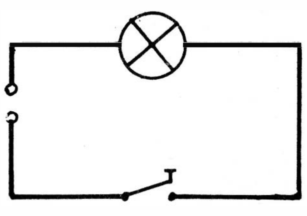
Nachdem wir uns im letzten Kapitel mit der Struktur von Assembleranweisungen beschäftigt haben und dabei auch schon die erste Assemblerbefehle gelernt haben, wollen wir in diesem Kapitel ein paar grundlegende Worte über die interne Arbeitsweise des Computers sagen. Wie Sie bereits wissen, ist Assembler eine maschinennahe Programmiersprache und es ist daher unerläßlich, sich ausführlich mit der „Maschine" auszukennen, als das zum Beispiel bei der BASIC-Programmierung notwendig ist.

Als wir im vorangegangenen Kapitel besprochen haben, daß der Computerspeicher aus einer Vielzahl von einzelnen Zellen besteht, haben Sie vielleicht angenommen, daß diese eine Art „kleinste Einheit" des Computers darstellen – daß sozusagen alles aus Zellen aufgebaut ist. Nun, das wäre nicht ganz richtig – aber auch nicht ganz falsch.

Unser Computer ist ein elektrisches Gerät. Und wie bei jedem elektrischen Gerät unterscheidet man zunächst einmal zwischen „Strom ein" und „Strom aus". Damit ist nun keineswegs gemeint, ob Ihr Computer ein- oder ausgeschaltet ist. Vielmehr befindet sich innerhalb des Computers eine Vielzahl von elektronischen Schaltern – insgesamt sind es über eine halbe Million Schalter. Und wenn wir nun in Assembler oder auch BASIC programmieren, dann steuern wir damit im Grunde nur diese Schalter oder wie man auch sagt „Schaltelemente".



Schalter geschlossen,
es fließt Strom



Schalter offen,
es fließt kein Strom

Natürlich befinden sich im Computer keine solchen mechanischen Schalter, die eine Lampe ein- oder ausschalten, sondern die Schaltelemente befinden sich in IC-Bausteinen („Chips"). Das Prinzip ist aber exakt dasselbe.

Die kleinste Einheit in jedem Computer ist also ein Schaltelement, das offen oder geschlossen sein kann. Der Fachmann sagt, das Element kann zwei „Zustände" annehmen und nennt diese beiden Zustände einfach „0" und„1". Und genau das wollen wir in Zukunft auch tun. Die „0" steht für „Schalter offen", die „1" für „Schalter geschlossen. Man nennt ein solches System auch „binär", d.h. „zweier Zustände fähig" und spricht von einem „Binärsystem".

Computerfachleute nennen einen einzelnen Schalter „Bit“, abgeleitet von „binary digit“, „Binärziffer“. Ein Bit ist also die Entscheidung zwischen „Strom ein“ und „Strom aus“, zwischen „9“ und „1“.

Der für uns hier interessante 6502-Mikroprozessor ist derart aufgebaut, daß intern jeweils genau 8 Schalter, d.h. 8 Bits, zusammengehören. Eine solche Einheit aus 8 Bits nennen wir „Byte“, ein Begriff, den Sie bestimmt schon einmal gehört haben. Ein Beispiel für ein Byte ist „00101110“.

1 Byte = (z.B.)	0	0	1	0	1	1	1	0
Bit Nummer	7	6	5	4	3	2	1	0

Wie Sie sehen, werden die einzelnen Bits eines Bytes von rechts nach links von 0 bis 7 durchnumeriert. Die Numerierung von rechts nach links hängt mit der sog. „Stellenwertigkeit“ zusammen, wir kommen noch darauf zurück.

Machen Sie sich klar, daß der Computer in seinem Innersten immer nur mit Bits und Bytes umgeht, gleichgültig wie großartig seine Leistungen nach außen hin erscheinen mögen. Für uns wirft das natürlich die Frage auf, wie der Computer mit diesen einfachsten Methoden komplexe Aufgaben lösen kann, ja, wie überhaupt Zahlen oder gar Zeichen damit gehandhabt werden können.

Nun, wie Sie bereits wissen, ist der Computerspeicher aus einzelnen Speicherzellen aufgebaut. Eine solche Zelle ist nun genau ein Byte groß. Man kann sich das ungefähr so vorstellen:

Speicherzelle 0:	0	1	0	1	0	1	0	1	= 8 Bit = 1 Byte
Speicherzelle 1:	0	1	1	1	0	0	0	1	
Speicherzelle 2:	1	1	1	0	1	1	0	0	
.									
.									
.									
Speicherzelle 65535:	0	1	0	0	0	1	1	0	

Der Speicher beinhaltet also eine „Kette“ von Zellen, die jeweils aus einem Byte bestehen. Die Bytes unterscheiden sich voneinander dadurch, daß sie zum Teil unterschiedliche Bitkombinationen beinhalten, zum Beispiel, Speicherzelle 0 hat „01010101“ Zelle 1 hingegen „01110001“ usw. Man redet auch davon, daß ein Bit „gesetzt“ (= 1) oder „gelöscht“ (= 0) ist. Die in der Skizze genannten Bitkombinationen sind willkürlich gewählt.

Es ist klar, daß es bei acht Bits nur eine begrenzte Anzahl unterschiedlicher Bitkombinationen gibt.

Kombination 0:	0	0	0	0	0	0	0	0
Kombination 1:	0	0	0	0	0	0	0	1
Kombination 2:	0	0	0	0	0	0	1	0
Kombination 3:	0	0	0	0	0	0	1	1
Kombination 4:	0	0	0	0	0	1	0	0
Kombination 5:	0	0	0	0	0	1	0	1
Kombination 6:	0	0	0	0	0	1	1	0
Kombination 7:	0	0	0	0	0	1	1	1
Kombination 8:	0	0	0	0	1	0	0	0
Kombination 9:	0	0	0	0	1	0	0	1
Kombination 10:	0	0	0	0	1	0	1	0
Kombination 11:	0	0	0	0	1	0	1	1
Kombination 12:	0	0	0	0	1	1	0	0
Kombination 13:	0	0	0	0	1	1	0	1
Kombination 14:	0	0	0	0	1	1	1	0
Kombination 15:	0	0	0	0	1	1	1	1
Kombination 16:	0	0	0	1	0	0	0	0
.								
.								
.								
Kombination 255:	1	1	1	1	1	1	1	1

Sicher ist Ihnen jetzt auch klar, warum wir mit dem 6502-Mikroprozessor grundsätzlich nur Zahlen zwischen 0 und 255 verarbeiten können. Andere Werte können weder vom Prozessor intern gehandhabt noch im Speicher abgelegt werden. Schließlich besteht eine einzelne Speicherzelle nur aus einem einzigen Byte. Allerdings ist es möglich, zwei hintereinander liegende Speicherzellen unter bestimmten Bedingungen als logische Einheit zu betrachten. Genau dies geschieht nämlich, wenn wir die Zellen durchnumerieren, d.h. ihnen Adressen geben.

Schließlich gibt es nicht nur 256 (0 bis 255) verschiedene Adressen, sondern genau 65536 (von 0 bis 65535). Die Zahl 65536 kommt zustande als Ergebnis von 256 mal 256. Rechnen Sie es ruhig nach! Zur Darstellung einer Adresse benötigt unser Computer also zwei Bytes (256 x 256 Bitkombinationen).

Adressen sind die einzigen Zwei-Byte-Zahlen die der 6502-Prozessor verarbeiten kann. Ansonsten steht nur immer jeweils ein Byte zur Verfügung. So können auch der Akkumulator sowie die X- und Y-Register nur ein einziges Byte beinhalten. Es gibt andere (neuere) Mikroprozessoren, die grundsätzlich zwei oder sogar noch mehr Bytes auf einmal bearbeiten können.

Bytes dienen nicht nur als Argument für Assemblerbefehle oder als Adressen, sondern auch zur Darstellung von Zeichen. Unter dem Begriff „Zeichen“ verstehen wir hierbei Ziffern (0 bis 9), Buchstaben („a“ bis „z“ und „A“ bis „Z“), Sonderzeichen (Komma, Punkt usw.) sowie graphische Zeichen. Da es 256 verschiedene Bytes gibt, existieren auch genausoviele unterschiedliche Zeichen.

Die Umwandlung einer bestimmten Bitkombination in ein Zeichen und umgekehrt ist recht einfach. Wenn Sie ein Zeichen von der Tastatur eingeben, so wird diese direkt von der Tastatur in eine Folge von acht Bits umgesetzt. Zur Darstellung auf dem Bildschirm gibt es einen sog. „Zeichengenerator“ im Computer, der jeder Bitkombination ein bestimmtes Muster auf dem Bildschirm zuordnet.

Für die Zuordnung von Bitkombinationen zu Zeichen existieren eine Vielzahl von Vorschlägen. Der wohl weltweit wichtigste Standard ist der ASCII-Code. Die Abkürzung „ASCII“ steht für „American Standard Code for Information Interchange“, d.h. „Amerikanischer Standardcode für Informationsaustausch“.

Ein Auszug aus dem ASCII-Code:

0	1	0	0	0	0	0	1	steht für „A“
0	1	0	0	0	0	1	0	„B“
0	1	0	0	0	0	1	1	„C“

usw.

Die meisten Computerhersteller halten sich mehr oder weniger streng an die ASCII-Norm. Commodore zum Beispiel hat den ASCII-Code stark erweitert und weicht in einzelnen Spezifikationen auch davon ab. Dieser leicht abweichende Code trägt oft die Bezeichnung ASC-Code. Das rührt daher, daß diese Codierung vor allem in BASIC verfügbar ist und mit dem ASC-Befehl abgefragt werden kann.

Darüber hinaus gibt es noch den Bildschirmcode, oft auch BSC-Code genannt. Diese Zuordnung von Bitkombinationen zu bestimmten Zeichen kommt zur Anwendung, wenn direkt auf den Bildschirm zugegriffen wird. Direkt meint hier, nicht zum Beispiel über den BASIC-Befehl PRINT, sondern mittels POKE oder PEEK. Da wir in Assembler gar keinen PRINT-Befehl zur Verfügung haben, wird für uns bei den meisten Bildschirmoperationen der BSC-Code von speziellem Interesse sein. Wie Sie wissen, gibt es ja einen POKE-ähnlichen Befehl in Assembler, nämlich STA. Der BSC-Code stimmt teilweise mit dem ASCII- bzw. ASC-Code überein. Die Codes lassen sich leicht ineinander überführen.

Unabhängig von den beiden verschiedenen Codes ASC und BSC hängt bei den Commodore-Geräten die Darstellung auf dem Bildschirm auch davon ab, ob Graphik- oder Textmodus gewählt ist.

Nun wäre es recht umständlich, wollte man jedes Zeichen tatsächlich in der Binärdarstellung, also als Nullen und Einsen, in den Computer eingeben, Immerhin – in früheren Computerzeiten war es durchaus üblich, alle Daten auf diese Weise in den Computer zu bringen. Heutige Computer beinhalten jedoch Routinen, die es uns erlauben, Zeichen als normale Dezimalzahlen auszudrücken.

Mit anderen Worten: Jede Bitkombination entspricht einer ganz bestimmten Zahl.

binär	dezimal
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
.	
.	
.	
11111111	255

Achtung: Verwechseln Sie diese Gegenüberstellung von Binär- und Dezimalsystem nicht mit den Codierung der ASC- oder BSC-Codes. Die Dezimaldarstellung ist lediglich eine andere – leichtere – Schreibweise für die Binärform. Mit der Dezimalzahl 3 drücken wir aus, daß die 8 Schalter des betreffenden Bytes wie folgt offen oder geschlossen sind: „00000011“. Darüber, wie diese Schaltereinstellung (Bitkombination) auf dem Bildschirm dargestellt wird, sagt die Zahl 3 nichts aus; kann sie ja auch nicht, denn wie wir wissen, ist das je nach Code verschieden.

Die Dezimalzahlen benutzen wir, wenn wir zum Beispiel „POKE 5,3“ eingeben. Wir ersparen uns damit „POKE ’00000101’, ’00000011’“. Da letzteres sehr umständlich wäre, ist es bei praktisch allen Mikrocomputern erst gar nicht zugelassen. In ’T.EX.AS.’ allerdings ist eine binäre Eingabe sehr wohl möglich, da sie bei Assemblerprogrammen hin und wieder sinnvoll ist.

Wir wollen an dieser Stelle eine kurze mathematische Begründung dafür geben, daß wir jede Binärzahl auch als Dezimalzahl schreiben können. Versuchen Sie dieser Erklärung soweit zu folgen, wie es Ihre mathematischen Kenntnisse zulassen. Für die Programmierung in Assembler reichen durchaus die bisher gemachten Ausführungen, verzweifeln Sie also nicht, wenn Sie die folgenden Erklärungen nicht vollständig verstehen. Sie erscheinen uns aber immerhin so interessant, daß wir sie dem interessierten Leserkreis nicht vorenthalten wollen.

Man kann ein binäres System nicht nur als eine Kombination von Schaltern verstehen, sondern auch als ein Zahlensystem, das aus nur zwei unterschiedlichen Ziffern, nämlich 0 und 1, besteht. Dieses System ist in seinen Grundstrukturen unserem Dezimalsystem mit seinen 10 Ziffern 0 bis 9 sehr ähnlich. Wir wollen uns deswegen den Aufbau des von uns üblicherweise benutzten Dezimalsystems einmal genauer ansehen.

Im Dezimalsystem existieren 10 verschiedene Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Indem wir beliebig viele Ziffern aneinanderhängen, können wir beliebige Zahlen darstellen. So besteht die Zahl „Einhundertachtundzwanzig“ aus der Ziffernfolge „128. Die Position einer Ziffer innerhalb einer Zahl bezeichnet man als „Stelle“. Jede Stelle hat ihren bestimmten Wert, die Ziffer an dieser Stelle bestimmt lediglich, wie oft dieser Wert vorkommt. Die Zahl 128 setzt sich zusammen aus 8 mal 1 plus 2 mal 10 plus 1 mal 100. Von rechts gelesen hat die erste Stelle also den Wert 1, die zweite den Wert 10, die dritte den Wert 100 usw. Jede Stelle trägt den Wert eines Vielfachen oder Teiles von 10. Daher nennen wir dieses uns so vertraute Zahlensystem auch „Zehner- oder Dezimalsystem“. Die Zahl 10 kommt daher, daß wir mit 10 Ziffern rechnen.

Der Mathematiker schreibt diese Vielfachen bzw. Teile von 10 auch als sog. „Potenzen“.

1	2	8
= 1 x 100	+ 2 x 10	+ 8 x 1
= 1 x 10 x 10	+ 2 x 10	+ 8 x 1
= 1 x 10↑2	+ 2 x 10↑1	+ 8 x 10↑0

Das Zeichen „ “ wird gelesen „hoch“ und bedeutet, daß die zuerst genannte Zahl so oft mit sich selbst multipliziert wird, wie es die zweite Zahl angibt.

Beispiele:
„10 hoch 5“ (10 ↑ 5) ist gleich 10 × 10 × 10 × 10 × 10
„2 hoch 4“ (2 ↑ 4) ist gleich 2 × 2 × 2 × 2
Jede Zahl (außer 0) hoch 0 ist definitionsgemäß gleich 1.
10 ↑ 0 = 2 ↑ 0 = 16 ↑ 0 = ... =1
Aber: 0 ↑ 0 = 0 (nicht 1)!

Die Tatsache, daß wir als Multiplikationszeichen hier ein „x“ benutzen, statt des in BASIC erforderlichen „*“, hat keine Bedeutung.

Zerlegen wir nun einmal die Zahl 3485 in ihre Potenzen:

```
3485 =    5 x 10 ↑ 0 =   5 x      1 =      5
        + 8 x 10 ↑ 1   + 8 x     10  +    80
        + 4 x 10 ↑ 2   + 4 x    100  +   400
        + 3 x 10 ↑ 3   + 3 x   1000  +  3000
                                     -------
                                      3485
```

Gehen wir nun vom Dezimalsystem über zum Binärsystem (auch „Dual- oder Zweiersystem" genannt). Eine Binärzahl, zum Beispiel 10110111, besteht aus den beiden Ziffern 0 und 1, die aneinandergehängt werden, um beliebige Zahlen darzustellen. Wir werden hier im allgemeinen mit 8 Binärstellen (Dualstellen) rechnen, da dies genau einem Byte entspricht. Grundsätzlich ist das Dualsystem aber ebenso wie das Dezimalsystem geeignet, beliebig große Zahlen darzustellen.

Wenn wir nun die Dualzahl 10110111 in unser Dezimalsystem umrechnen wollen, so können wir dazu das oben gezeigte Schema benutzen. Allerdings müssen wir berücksichtigen, daß die Basis für die Potenzen nun nicht mehr 10, sondern vielmehr 2 ist. Schließlich haben wir es ja mit einem Zweiersystem zu tun.

```
10110111 =
              1 x 2 ↑ 0 = 1 x    1 =     1
            + 1 x 2 ↑ 1 = 1 x    2  +    2
            + 1 x 2 ↑ 2 = 1 x    4  +    4
            + 0 x 2 ↑ 3 = 0 x    8  +    0
            + 1 x 2 ↑ 4 = 1 x   16  +   16
            + 1 x 2 ↑ 5 = 1 x   32  +   32
            + 0 x 2 ↑ 6 = 0 x   64  +    0
            + 1 x 2 ↑ 7 = 1 x  128  +  128
                                   -------
                                    183
```

Sie wissen ja: „2 ↑ 4" ist gleich „2 x 2 x 2 x 2".

Die Dualzahl 10110111 entspricht also der Dezimalzahl 183. Man kann Dualzahlen übrigens auch addieren und subtrahieren, wir wollen aber an dieser Stelle darauf verzichten, das auch noch zu lernen.

Sie haben aber jetzt recht deutlich gesehen, daß die binäre Arbeitsweise eines Comupters ihre mathematischen Grundlagen hat. Und wenn man diese kennt, hat man natürlich schon ein paar Verständnisvorteile bei der Assemblerprogrammierung. Deswegen haben wir sie hier besprochen. Wenn Ihnen das alles noch ein bißchen fremd vorkommt, lesen Sie es einfach nochmals und rechnen Sie selbst ein paar Beispiele durch.

Wir wollen zum Abschluß dieses Kapitels noch den Beweis erbringen, daß die mit acht Binärziffern – also einem Byte – höchste darstellbare Zahl tatsächlich 255 ist. Dazu nehmen wir einfach wieder unser Schema zur Hand setzen alle acht Stellen auf 1:

```
11111111  =
                 1 x 2 ↑ 0  = 1 x   1 =   1→
               + 1 x 2 ↑ 1  = 1 x   2  +   2
               + 1 x 2 ↑ 2  = 1 x   4  +   4
               + 1 x 2 ↑ 3  = 1 x   8  +   8
               + 1 x 2 ↑ 4  = 1 x  16  +  16
               + 1 x 2 ↑ 5  = 1 x  32  +  32
               + 1 x 2 ↑ 6  = 1 x  64  +  64
               + 1 x 2 ↑ 7  = 1 x 128  + 128
                                        ─────
                                         255→
```

Zusammenfassung

Die kleinste Informationseinheit eines Computers ist ein Bit. Jedes Bit kann genau zwei Zustände einnehmen, abstrahiert als 0 und 1 bezeichnet. Ein solches System heißt Binär-, Dual- oder einfach Zweiersystem.

8 Bit zusammen bilden eine logische Einheit, ein Byte. Jede Speicherzelle des Computers entspricht genau einem Byte, ebenfalls die internen Register des Mikroprozessors (8-Bit Prozessor).

Ein Byte kann 256 verschiedene Bitkombinationen darstellen, die nach mathematischen Regeln aus dem Binärsystem in unser Dezimalsystem umgewandelt werden können und die Zahlen von 0 bis 255 ergeben.

Diesen Zahlen können verschiedene Zeichen (Buchstaben etc.) zugeordnet werden. Die wichtigsten Zuordnungen sind der ASC und der BSC-Code.

Kapitel 4: Das zweite Programm – LDX, LDY, STX, STY, INX, INY, DEZ, DEY, INC, DEC.

Nachdem es im letzten Kapitel reichlich „trocken" zuging, wollen wir nun wieder aktiv programmieren. Je mehr Sie dabei mit der Assemblersprache vertraut werden, umso mehr werden Sie auch selbst experimentieren wollen. Dem steht nichts im Wege, beachten Sie aber unbedingt den folgenden Hinweis:

Bei Fehlern in BASIC bringt Ihnen der Computer eine Fehlermeldung und Sie können sich die betreffende BASIC-Zeile ansehen und korrigieren. Kein Problem. In Assembler bestehen solche konfortablen Möglichkeiten nicht. Daran kann auch 'T.EX.AS.' nicht viel ändern.

Ein Fehler in einem Assemblerprogramm bewirkt beim Ablauf im allgemeinen ein völlig unkontrolliertes Verhalten des Mikroprozessors und führt meist zu einem absoluten Datenverlust. Wenn Sie das Programm mit dem 'T.EX.AS.'-Befehl „EX" gestartet haben, hilft manchmal noch das Drücken der STOP-Taste, aber auch diese Methode funktioniert nicht immer. Bei einem Start des Programms von BASIC aus mit SYS besteht nicht einmal diese Chance. Wenn Ihr Computer eine Reset-Möglichkeit hat, hilft unter Umständen diese. Beim VC-20 und Commodore 64 kann zum Beispiel ein Rücksetzen des Prozessors durch gleichzeitiges Drücken von RESTORE und STOP bewirkt werden, beim Apple mittels Reset-Taste.

Sie sollten jedoch in der Entwicklungsphase eines Assemblerprogramms bei jedem Start des Programms oder eines Teils damit rechnen, daß sich ein Fehler eingeschlichen hat, der den Mikroprozessor in solch einen unkontrollierbaren Zustand bringt. In so einem Fall hilft nur noch Aus- und Wiedereinschalten des Computers. Lassen Sie das Gerät einige Sekunden ausgeschaltet, damit die fehlerhafte Information und alle anderen wirklich verloren geht.

Bei umfangreicheren Programmen ist es natürlich empfehlenswert, diese auf einer Diskette oder Kassette abzuspeichern, B E V O R Sie Ihr Programm starten, um es auszuprobieren. Das Abspeichern kostet zwar ein bißchen zusätzliche Zeit, aber selbst die längste Abspeicherung dauert nicht so lange, wie Sie für ein erneutes Eingeben oder gar Neuentwickeln des Programms benötigen.

DAHER: ERST SICHERN, DANN STARTEN.

So, jetzt kommen wir endlich zur Praxis. Wir haben bereits gelernt, ein einzelnen Zeichen auf dem Bildschirm auszugeben. Warum sollte diese Methode nicht auch bei kompletten Texten funktionieren?

Sie erinnern sich, mit LDA# haben wir den entsprechenden Code in den Akkumulator geladen, um ihn danach mittels STA in einer Speicherzelle des Bildschirms abzulegen. Dabei mußten wir nur die unterschiedlichen Adressen der verschiedenen Computertypen beachten.

Wir wollen jetzt ein Programm schreiben, das den Text „TEXAS“ auf dem Bildschirm ausgibt. Das hier wiedergegebene Programm bezieht sich beispielhaft auf Commodore CBM Computer (Bildschirmadressen 32768, 32769, 32770 ...). Die Umstellung für andere Computertypen wie Commodore 64, VC-20, Apple etc. ist aber völlig unproblematisch. Schlagen Sie einfach die Anfangsadresse des Bildschirms auf Seite 2-2 nach.

Unser neues Programm soll wiederum bei Adresse 5000 beginnen. Während wir bei BASIC normalerweise das vorherige Programm mit NEW löschen, ist dies bei Assembler nicht erforderlich. Wir schreiben unsere neuen Assembleranweisungen einfach „über“ die früheren drüber.

Und so sieht unser Programm aus:

CBM (Bildschirmanfang bei 32768, ggf. ändern):

```
5000 LDA    #20   ;BSC-Code für „T“
5002 STA  32768   ;32768=Bildschirmanfangsadresse
5005 LDA     #5   ;„E“
5007 STA  32769   ;Bildschirmanfangsadresse+1
5010 LDA    #24   ;„X“
5012 STA  32770   ;Bildschirmanfangsadresse+2
5015 LDA     #1   ;„A“
5017 STA  32771   ;Bildschirmanfangsadresse+3
5020 LDA    #19   ;„S“
5022 STA  32772   ;Bildschirmanfangsadresse+4
5025 RTS          ;beendet den Programmablauf
```

Die Eingabe mittels 'T.EX.AS.' oder einem anderen von ihnen benutzten Assemblersystem sollte Ihnen keine Schwierigkeiten mehr bereiten. Wenn Sie einen anderen Computertyp benutzen, vergessen Sie auf keinen Fall, die fünf Adressen 32768, 32769, 32770, 32771 und 32772 durch die für Ihren Computer gültigen Werte zu ersetzen. Andernfalls werden Sie bei einem Programmlauf nicht nur nichts sehen, sondern u.U. auch ein „Datenchaos“ im Speicher auslösen. Beachten Sie außerdem, daß die jeweiligen Kommentare, die mit Semikolon hinter jedem Assemblerbefehl stehen, nicht eingegeben werden dürfen. Sie dienen nur zur Erklärung.

Doch bevor wir dazu kommen, wollen wir das Programm erst einmal ausprobieren. Wie Sie bereits von unserem ersten Programm her wissen, müssen wir dazu bei einigen Computermodellen zunächst die Bildschirmfarbe verändern. Dies ist notwendig, da andernfalls unser Text „TEXAS“ in derselben Farbe ausgegeben würde, die der Bildschirm sowieso schon hat. Und dann ist natürlich nichts lesbar oder haben Sie schon einmal probiert, schwarze Schrift auf schwarzem Grund zu enträtseln? Schlagen Sie also erforderlichenfalls noch einmal auf Seite 2-3 nach, wie die Farbe des Bildschirms bei Ihrem Computer geändert werden kann (speziell bei Commodore 64 und VC-20).

Starten Sie nun unser Programm von BASIC aus mit ...

SYS 5000 bezw. CALL 5000

.. oder von 'T.EX.AS.' aus mit ...

EX 5000

Wenn Sie alles richtig gemacht haben, wird das Wort „texas" oder „TEXAS" in der linken oberen Ecke des Bildschirms ausgegeben. Die Darstellung groß oder klein hängt davon ab, ob der Computer in Graphik- oder Textmodus geschaltet ist.

Die Erklärung für das Programm? Nun, die zugehörigen Kommentare zusammen mit Ihrem bereits erworbenen Vorwissen sollten eigentlich ausreichen, um das Programm zu verstehen. Folgende Anmerkungen dazu werden das Gesamtbild abrunden:

STA benötigt den BSC-Code eines Zeichens zur Darstellung auf dem Bildschirm. In diesem Code existieren zwei Alphabete:

BSC-Code (variiert von Computer zu Computer, probieren Sie!):

1 ... 26 entspricht a ... z bzw. A ... Z
65 ... 90 entspricht A ... Z bzw. Graphikzeichen

Die erste Angabe nach „entspricht" gilt für den Textmodus (Groß- und Kleinschreibung), die zweite für den Graphikmodus.
Während wir in unserem ersten Beispiel den Code 65 verwendet und dabei entweder ein großes .„A" oder das Graphikzeichen „Pik" erzielt hatten, wird diesmal das Wort „TEXAS" im Codebereich von 1 bis 26 geschrieben.

Die Vorgehensweise ist dieselbe wie schon in unserem ersten Programm. Mittels „LDA #" wird der gewünschte Code in den Akkumulator geladen, um anschließend mit „STA" an die entsprechende Bildschirmstelle gespeichert zu werden. Es gehören also immer ein LDA- und ein STA-Befehl zusammen. Der erste lädt den Code, der zweite gibt ihn auf dem Bildschirm aus.

Wir möchten Sie an dieser Stelle ermuntern, jetzt einmal dieses Buch hier aus der Hand zu legen und selbständig zu versuchen, das Programm derart zu ändern, daß statt „TEXAS" nun „T.EX.AS." ausgegeben wird wie Sie wissen, als Abkürzung für „Terminal Extended Assembler". Der BSC-Code für den Punkt ist 46.

Zur Vorgehensweise: Schreiben Sie das Programm ab Adresse 5000 am besten noch einmal völlig neu, fügen aber an den entsprechenden Stellen die neuen LDA- und STA-Anweisungen für die Punkte ein. Bedenken Sie, daß sich dadurch natürlich auch die Bildschirmadressen für die anderen nachfolgenden STA-Befehle erhöhen. Vergessen Sie nicht das RTS am Ende!

Um die Adressen, an denen das Programm steht, sollten Sie sich im Moment noch gar nicht weiter kümmern. Beginnen Sie mit der Eingabe des ersten Befehls, der sich ja nicht verändert, bei Adresse 5000. 'T.EX.AS.' gibt die richtigen Folgeadressen automatisch aus. Wir werden zu einem späteren Zeitpunkt besprechen, warum die Adressabstände zwischen den einzelnen Befehlen unterschiedlich sind. Im wesentlichen hängt das mit der Länge der Anweisungen zusammen.

So, nun aber los! Wann schreibt Ihr Programm „T.EX.AS". mit Punkten? Vorher sollten Sie nicht mit dem nächsten Abschnitt beginnen.

Wie wir bereits erwähnt haben, existieren im Mikroprozessor außer dem Akkumulator noch zwei andere interne Register, wir haben sie X- und Y-Register genannt. Diese Register sind dem Akkumulator sehr ähnlich, auch sie können Zahlen zwischen 0 und 255 jeweils einschließlich beinhalten. Für uns bedeutet das, daß wir unser Textprogramm problemlos auch zum Beispiel mit dem X-Register schreiben können. Dann sieht es so aus:

CBM (für andere Computer Bildschirmadressen ändern):

```
5000 LDX    #20      Vergleichen Sie mit
5002 STX  32768      dem entsprechenden
5005 LDX     #5      Programm auf Seite
5007 STX  32769      4 –2.
5010 LDX    #24
5012 STX  32770
5015 LDX     #1
5017 STX  32771
5020 LDX    #19
5022 STX  32772
5025 RTS
```

Es dürfte Ihnen nicht schwerfallen, auch dieses Programm zu verstehen. Statt LDA kommt LDX und statt STA nun STX zur Anwendung. Während das „A" für Akkumulator steht, bedeutet das „X" einfach X-Register. Unser Vergleich mit BASIC sieht so aus:

LDX #5	..entsprichtetwa...	X = 5
STX 32768	..entsprichtetwa...	POKE 32768, X

LDX bedeutet also „load x-register", „lade X-Register", STX steht für „store x-register", „speichere X-Register ab". Ähnlich wie bei STA im Akkumulator bleibt bei STX der Wert weiterhin im X-Register erhalten. Es wird lediglich ein Duplikat an die angegebene Speicherzelle gespeichert.

Wenn wir schon bei den internen Registern des Prozessors sind, wollen wir der Vollständigkeit halber auch noch gleich das Y-Register besprechen. Die für uns im Augenblick wichtigen Befehle dazu lauten LDY und STY, die Bedeutung ist entsprechend den ähnlich lautenden Anweisungen für das X-Register.

```
LDY  #5      .. entspricht etwa ...   Y=5

STY  32768   .. entspricht etwa ...   POKE 32768,Y
```

Auch wenn es Ihnen jetzt so erscheint, als ob Akkumulator, X- und Y-Register völlig gleich seien, so stimmt das nur bedingt. Es gibt eine ganze Reihe von Funktionen, die nur von jeweils einem der drei Register geleistet werden können.

Wir wollen uns jetzt gleich einmal zwei Funktionen ansehen, die nur mit dem X-oder Y-Register möglich sind, nicht jedoch mit dem Akkumulator. Beginnen wir mit dem Befehl INX. Er bezieht sich auf das X-Register.

Bevor wir INX erklären, wollen wir die Wirkung ausprobieren. Geben Sie dazu einmal das folgende kurze Programm ein:

CBM (für andere Computer andere Bildschirmadressen!):

```
5000 LDX    #1      ;X-Register mit 1 laden (BSC: „A“)
5002 STX  32768     ;an erster Bildschirmpos. speichern
5005 INX            ;X-Register um eins erhöhen („B“)
5006 STX  32769     ;an zweiter Bildschirmpos. speichern
5009 RTS
```

Probieren Sie das Programm aus. Daß Sie dazu bei anderen Computern als Commodore CBM die Bildschirmadressen (32768 und 32769) anpassen sowie ggf. die Bildschirmfarbe verändern müssen, wollen wir an dieser Stelle zum letzten Mal erwähnen. Schließlich sind Sie ja mittlerweile kein Anfänger mehr. Der Start des Programm kann – wie üblich – von BASIC aus mit „SYS 5000“ bezw. „CALL 5000“ oder in 'T.EX.AS.' mit „EX 5000“ erfolgen.

Das Programm druckt die beiden Buchstaben „A“ und „B“ nebeneinander auf den Bildschirm. Wie geschieht das nun im einzelnen?

Als erstes wird das X-Register mit dem Wert 1 geladen. Dies entspricht im BSC-Code dem Buchstaben „A“ bzw. „a“ (Graphik- oder Textmodus). Mit der ersten STX-Anweisung wird der Buchstabe an der ersten Bildschirmadresse – im Beispiel 32768 – abgespeichert. Soweit ist das nicht neu für uns. Der nun folgende Befehl INX steht für „increment x-register“, zu deutsch „erhöhe X-Register“, und zwar um genau eins. Aus der 1 im X-Register wird also eine 2.

Im Vergleich mit BASIC könnten wir für INX schreiben:

INX .. entspricht etwa ... X = X+1

Der Wert im X-Register wird also durch INX um eins erhöht. Beachten Sie dabei: Das X-Register kann – wie Sie wissen – nur Werte von 0 bis 255 annehmen. Wenn nun 255 im X-Register steht und Sie mit INX probieren, den Wert noch weiter zu erhöhen, so beginnt das Register wieder bei 0.

In unserem Beispielprogramm bewirkt INX, daß aus der 1 im X-Register eine 2 wird. Das aber ist der BSC-Code für den Buchstaben „B“ oder „b“. Die nachfolgende STX-Anweisung an die zweite Bildschirmadresse (im Beispiel 32769) stellt also diesen Buchstaben auf dem Bildschirm dar. Die abschließende RTS-Anweisung entsprechend dem END-Befehl in BASIC ist für uns bereits zur Selbstverständlichkeit geworden.

Machen Sie sich klar, daß unser Programm nur funktioniert, da der erste STX-Befehl nur eine Kopie der 1 in Speicherzelle 32768 ablegt. Die 1 bleibt trotzdem noch im Register selbst erhalten – andernfalls könnte sie ja nicht direkt danach um eins erhöht werden.

Unsere Assemblersprache bietet uns nicht nur eine Möglichkeit zum Erhöhen des X-Registers um eins, sondern auch zum Erniedrigen um eins. Der dafür zuständige Befehl lautet DEX. „DEX“ steht für „decremet x-register“, „erniedrige das X-Register um eins“.

DEX ... entspricht etwa .. X = X-1

Schreiben wir auch hierfür ein kleines Programm:

```
5000 LDX     #2       ;X-Register mit BSC-Code „B“ laden
5002 STX  32768       ;an erster Bildschirmpos. speichern
5005 DEX              ;X-Register um eins erniedrigen („A“)
5006 STX  32769       ;an zweiter Bildschirmpos. speichern
5009 RTS              ;Programmende
```

Tippen Sie das Programm ein und probieren Sie es aus. Ändern Sie doch einmal die 2 in eine andere Zahl oder vertauschen die beiden STX-Anweisungen. Was passiert, wenn wir DEX durch INX ersetzen? Wie ist es mit den uns bekannten Mitteln in Assembler auf einfache Weise möglich, das gesamte Alphabet auf dem Bildschirm darzustellen? Nehmen Sie sich die Zeit, mit den neuen Befehlen ein bißchen zu experimentieren. Je mehr Sie sich jetzt „spielerisch“ mit den Assemblermöglichkeiten befassen, umso besser werden Sie später zurechtkommen, wenn Sie einmal ein komplexes Assemlerprogramm entwickeln wollen.

Wenn Sie den Rat befolgt und tatsächlich selber experimentiert haben, dann wissen Sie jetzt vielleicht auch schon, was passiert, wenn das X-Register den Wert 0 beinhaltet und dennoch mit DEX probiert wird, das Register um eins zu erniedrigen. Die Lösung ist analog der von INX: Aus der 0 wird eine 255.

Nun, nachdem Sie jetzt mit INX und DEX bestens vertraut sind, werden Ihnen auch die Assemblerbefehle INY und DEY keine Probleme mehr bereiten. Es sind die entsprechenden Anweisungen zum Erhöhen bzw. Erniedrigen des Y-Registers.

INY ... entspricht etwa .. Y=Y+1

DEY ... entspricht etwa .. Y=Y-1

Ein analoger Befehl für den Akkumulator existiert nicht. Aber wir haben ja bereits festgestellt, daß die drei internen Register des Prozessors – obwohl in ihrem Aufbau identisch – nicht in allen Funktionen gleich sind. Auch zwischen dem X- und Y-Register bestehen einige gravierende Unterschiede, die uns aber erst zu einem späteren Zeitpunkt interessieren sollen.

Neben den Anweisungen zum Erhöhen und Erniedrigen des X- und Y-Registers gibt es auch Befehle, die diese Operationen innerhalb einer bestimmten Speicherzelle ermöglichen. Und damit sind wir zum ersten Mal bei Assemblerbefehlen, die komplexere Funktionen erfüllen.
Das Befehlswort zum Erhöhen einer Speicherzelle lautet „INC“ für „increment“, zum Erniedrigen „DEC“, „decrement“. Während zum Beispiel DEX sofort klarmacht, daß hier das X-Register erniedrigt wird, muß zu INC und DEC noch eine zusätzliche Adresse angegeben werden. Schließlich muß ja festgelegt werden, der Inhalt welcher Speicherzelle eins hoch- oder heruntergezählt werden soll.

Die Anweisung ...

INC 32768

.. erhöht also den Inhalt der Speicherzelle 32768 um eins. Wenn dort vorher eine 1 stand, wird es nach dieser Anweisung eine 2 sein.

Das klingt relativ einfach. Wenn man sich aber ansieht, wie der Prozessor diesen Befehl verarbeitet, dann wird klar, warum er zu den komplexeren gehört. Eine Berechnung der Form „erhöhe um eins“ kann nämlich nur innerhalb des Prozessors selbst stattfinden, niemals in einer Speicherzelle. Wenn nun eine INC-Anweisung auftaucht, dann holt sich der Prozessor den Inhalt (Wert) dieser Speicherzelle in den internen Rechenteil, addiert eine 1 dazu und speichert anschließend das Ergebnis wieder in der Speicherzelle ab.

Daß dies etwas komplizierter ist, sehen Sie auch bei dem Vergleich mit BASIC.

INC 32768 ... entspricht etwa .. POKE 32768, PEEK (32768)+1

Der Inhalt der Speicherzelle wird „gelesen“ (PEEK), um eins erhöht (+1) und dann wieder zurückgespeichert (POKE). In Assembler ist dies alles mit einem einzigen Befehl, nämlich INC, möglich.

Am einfachsten sehen wir uns den neuen Befehl gleich einmal an einem Beispiel an.

```
5000 LDA    #1         ;Akku mit BSC-Code „A“ laden
5002 STA 32768         ;an erste Bildschirmpos. speichern
5005 INC 32768         ;erste Bildschirmpos. um eins erhöhen
5008 RTS               ;Programmende
```

Dieses Programm ist lediglich zu Lehrzwecken gedacht, kein erfahrener Programmierer würde es so benutzen. Uns genügt es aber, um zu demonstrieren, worum es geht.

Geben Sie das Programm ein und probieren Sie es aus. Das Ergebnis wird sein, daß an der ersten Bildschirmposition ein „B“ oder „b“ dargestellt wird. Überrascht Sie das? Nun, die Erklärung dafür ist jedenfalls recht einfach.

Mit den ersten beiden Anweisungen (LDA und STA) wird ein „A“ auf den Bildschirm gebracht. In der entsprechenden Speicherzelle (32768 bei CBM-Computern) steht also eine 1. Wenn nun der INC-Befehl diese 1 um eins auf 2 erhöht, so erscheint in diesem Augenblick ein „B“ auf dem Bildschirm. Der Wechsel von „A“ nach „B“ geht so schnell vor sich, daß das „A“ gar nicht erst sichtbar wird.

Zu DEC zum Erniedrigen einer Speicherzelle wollen wir uns ebenfalls ein Beispiel ansehen:

```
5000 LDA    #1
5002 STA 32768
5005 DEC 32768
5008 RTS
```

Das Programm ist nicht mit Kommentaren versehen, aber das können Sie ja sicherlich selbständig nachholen. Auch wenn es einfach zu verstehen ist – ausprobieren sollten Sie aber das Programm dennoch. Das ausgegebene Zeichen heißt korrekt „AT-Zeichen“, wird aber im Fachjargon meist nur „Klammeraffe“ genannt.

Ziehen wir auch für DEC den Vergleich mit BASIC:

DEC 32768...entspricht etwa ..POKE 32768, PEEK (32768)-1

Für INC und DEC gilt – genau wie für INX, DEX, INY und DEY – daß bei Über- oder Unterschreiten des Bereichs von 0 bis 255 das Register jeweils wieder von vorne bzw. hinten beginnt zu zählen. Nach der 255 folgt also wieder die 0.

So, damit wollen wir den Lehrstoff für diesmal abschließen. Immerhin haben Sie in diesem Kapitel eine ganze Reihe von neuen Assemblerbefehlen kennengelernt. Wir wollen Sie nochmals zum eigenen Experimentieren damit ermuntern. Und lassen Sie sich nicht entmutigen, wenn Sie noch den einen oder anderen Fehler machen und der Computer dabei u.U. funktionsuntüchtig wird. Durch Aus= und Wiedereinschalten können Sie ihn in jedem Fall wieder „reparieren".

Zusammenfassung

Ein fehlerhaftes Assemblerprogramm kann zu völligem Datenverlust führen.

Mit den Befehlen LDA und STA kann ein Text auf dem Bildschirm ausgegeben werden. Die entsprechenden Befehle für das X=Register lauten LDX und STX, für das Y-Register LDY und STY. (LDA und STA haben wir in Kapitel 2 besprochen.)

```
LDX #argument          ;lädt X-Register mit Argument (0..255)
STX adresse            ;speichert X-Reg. an der Adresse ab
LDY #argument          ;lädt Y-Register mit Argument (0..255)
STY adresse            ;speichert Y-Reg. an der Adresse ab
```

Assembler bietet Befehle zum Erhöhen und Erniedrigen von Registern und Speicherzellen.

```
INX                    ;erhöht X-Register um eins
DEX                    ;erniedrigt X-Register um eins
INY                    ;erhöht Y-Register um eins
DEY                    ;erniedrigt Y-Register um eins
INC  adresse           ;erhöht Inhalt der Speicherzelle
DEC  adresse           ;erniedrigt Inhalt der Speicherzelle
```

Beim Dekrementieren (Erniedrigen) ist der auf 0 folgende Wert 255, beim Inkrementieren (Erhöhen) der auf 255 folgende Wert 0.

Ein IN? oder DE? entsprechender Befehl für den Akkumulator existiert nicht.

Kapitel 5: Speicheraufbau

Wir haben bereits darüber gesprochen, was wir in Assembler unter einer Adresse zu verstehen haben und haben auch schon praktische Erfahrungen damit gemacht. Dennoch gibt es hierzu noch einiges nachzutragen, was wir jetzt tun wollen.

Wie Sie wissen, ist der gesamte Speicher des Computers aus einzelnen Speicherzellen aufgebaut. Diese Zellen haben unterschiedliche Aufgaben. Wir wollen uns hier beispielhaft einmal den Speicheraufbau eines Commodore CBM 8032 ansehen:

Adresse		
65535		größte Adresse
	Betriebssystem, BASIC-Interpreter, Ein und Ausgaberoutinen (ROM-Bereich)	(beinhaltet praktisch die „Intelligenz“ des Computers)
36864		
	Bildschirm-RAM	(wird je nach Computertyp nur zum Teil ausgenutzt)
32768		
	Programm- und Variablenspeicher (freie RAM-Kapazität)	Das Betriebssystem teilt diesen Bereich nochmals in sich auf: Programmtext, einfache Variablen, Stringvariablen, Feldvariablen (Arrays).
1024		
	Betriebssystem-zwischenspeicher	(beinhaltet u.a. „Stack“ und „Zero-Page“)
0		

Die zwei für uns neuen Begriffe „RAM“ und „ROM“ sind schnell erklärt. Ein RAM-Speicher ist derart aufgebaut, daß man in ihn jederzeit Daten einspeichern kann, die zu einem späteren Zeitpunkt wieder abrufbar sind. Das ist zum Beispiel mit jedem normalen BASIC-Programm der Fall: Sie geben es ein und können es hinterher mit LIST zurückrufen oder mit RUN ausführen. Wenn Sie aber den Strom abschalten, gehen die Informationen verloren. Hingegen sind Betriebssystem und BASIC-Interpreter beim CBM 8032 und vielen anderen Computern in ROM-Speichern abgelegt. Ein ROM behält seinen Inhalt auch bei abgeschaltetem Strom. Daher versteht der Computer direkt nach dem Einschalten BASIC. Der Inhalt eines ROM-Speichers kann – im Gegensatz zu einem RAM – nicht geändert werden.

ROM-Speicherzellen können also nur „gelesen“ werden, während RAM-Speicherzellen „beschrieben“ und „gelesen“ werden können. Dies ist der einzige Unterschied. „ROM“ steht übrigens für „read only memory“ (Nur-Lesespeicher), „RAM“ für „random access memory“ (Wahlfreier Zugriffsspeicher).

Und da wir schon gerade bei Fachausdrücken sind, statt „Ein-/Ausgaberoutinen“ wird häufig einfach „I/0“ oder schlicht „I0“ gesagt, als Abkürzung für „input/output“, „Ein-/Ausgabe“.

Schließlich gehört zu einem fortgeschrittenen Programmierer auch die Kenntnis der entsprechenden Fachtermini.

Abgesehen davon zeigt uns das Schaubild, in welchen Adressbereichen welche Funktionen des Computers liegen. Machen Sie sich klar, daß es sich dabei letztlich immer nur um Speicherzellen handelt, die eine Zahl von 0 bis 255 beinhalten womit wir ein ganz wesentliches Prinzip aller heutigen Mikrocomputer ansprechen.

Das Prinzip lautet: Es gibt keinen grundsätzlichen Unterschied zwischen Daten und Programmen. Intern wird alles durch Zahlen dargestellt.

Auch unser Assemblerprogramm oder ein BASIC- oder PASCAL-Programm wird im Computer als eine Folge von Zahlen in Speicherzellen abgelegt. Dadurch, daß diese Zahlenketten in ganz bestimmter Weise interpretiert werden, stellen sie ein Programm dar. Wenn wir ein Programm in Assembler oder BASIC eintippen, hat das Assemblersystem oder der BASIC-Interpreter nichts eiligeres zu tun, als unser Programm Befehl für Befehl in Zahlen umzuwandeln und zu speichern. Wenn wir das Programm wieder auflisten wollen, wird es wieder in lesbare Zeichen zurückverwandelt. Das klingt recht aufwendig, aber so ist es nun einmal.

Zahlen, Rechenergebnisse, Strings usw. – also Daten – werden natürlich ebenfalls intern als Zahlen gespeichert. Und diesen Zahlen ist in keiner Weise anzusehen, ob sie Teil eines Programms oder zum Beispiel eines Strings sind.

In einer höheren Programmiersprache wie BASIC übernimmt der BASIC-Interpreter diese Unterscheidung, deswegen fällt dort die grundsätzliche Gleichheit von Programmen und Daten normalerweise gar nicht auf. In Assembler hingegen ist es Ihre Aufgabe als Programmierer, dafür zu sorgen, daß „Programmzahlen“ und „Datenzahlen“ nicht durcheinander geraten. Und das ist manchmal gar nicht so einfach.

Wie gesagt – in BASIC existiert das Problem überhaupt nicht. In Assembler aber müssen wir uns damit befassen, wie Assemblerbefehle als Zahlen im Speicher abgelegt werden. Keine Angst – so schwer ist das nicht. Wichtig ist zunächst, daß Sie alle in den vorangegangenen Absätzen besprochenen Grundprinzipien voll verstanden haben.

Sehen wir uns nun einmal an, wie ein Assemblerprogramm im Speicher als Zahlenfolge abgelegt wird. Wir nehmen dazu ein einfaches Beispiel:

5000	LDX	#65	.. wird gespeichert als ..	162	65	
5002	STX	32768	.. wird gespeichert als ..	142	0	128
5005	INX		.. wird gespeichert als ..	232		
5006	STX	32769	.. wird gespeichert als ..	142	1	128
5009	RTS		.. wird gespeichert als ..	96		

Jedem Assemblerprogrammschritt werden also ein bis drei Zahlen zugeordnet. Diese Zahlen werden hintereinander in den entsprechenden Speicherzellen abgelegt. Unser Beispielprogramm steht wie folgt im Speicher:

Adresse	Inhalt (dezimal)	Bedeutung
5000	162	LDX#
5001	65	Argument zu LDX#
5002	142	STX
5003	0	1. Teilargument zu STX
5004	128	2. Teilargument zu STX
5005	232	INX
5006	142	STX
5007	1	1. Teilargument zu STX
5008	128	2. Teilargument zu STX
5009	96	RTS

Wie Sie sehen, ist jedem Assemblerbefehlswort (LDX, STX, INX und RTS) ein bestimmter Code zugeordnet. Argumente im Bereich von 0 bis 255 werden direkt hinter dem Befehlswort abgespeichert (die 65 in unserem Beispiel). Größere Argumente wie 32768 und 32769 werden in zwei Bytes zerlegt (1. und 2. Teilargument), die nacheinander im Speicher stehen. Wie diese Aufteilung ein zwei Teilargumente vorgenommen wird, werden wir uns noch ansehen.

Wir unterscheiden in diesem Zusammenhang zwischen Ein-, Zwei- und Drei-Byte-Befehlen. Ein-Byte-Befehle sind solche, die kein Argument benötigen und daher nur ein einziges Byte im Speicher belegen, in unserem Programm also INX und RTS. Zwei-Byte-Befehlen benötigen zwei Bytes im Speicher – einen für das Befehlswort selbst, den anderen für das dahinterstehende Argument von 0 bis 255. In unserem Beispiel ist das der „LDX#65“-Befehl. Bei Drei-Byte-Befehlen schließlich ist das Argument größer als 255 und wird deshalb in zwei Teilargumente zerlegt, die hinter dem Befehlswort zwei weitere Bytes belegen. Jetzt wissen Sie auch, warum die Adressabstände zwischen den einzelnen Assemblerbefehlen (5000, 5002, 5005, 5006, 5009) unterschiedlich sind – die Befehle benötigen eine unterschiedliche Anzahl von Bytes im Speicher.

Die Zuordnung der Assemblerbefehlsworte zu bestimmten Zahlen finden Sie im Anhang dieses Buches. Für die meisten Befehlsworte existieren mehrere verschiedene Codes, die jeweils unterschiedliche Arten eines Befehls ausdrücken. So gibt es zum Beispiel den uns bekannten „LDA#“-Befehl, es gibt aber auch einen LDA-Befehl ohne „#“. Wir werden darauf noch zu sprechen kommen.

Jetzt wollen wir uns erst einmal darüber unterhalten, wie ein Argument oder überhaupt jede Zahl größer als 255 in zwei Teilzahlen zerlegt wird. Wie Sie wissen, ist das notwendig, da ein Byte (eine Speicherzelle) nur von 0 bis 255 reicht und größere Zahlen in zwei Bytes gespeichert werden müssen.

Nun kann man zum Beispiel die Zahl „32567“ nicht einfach in der Mitte trennen, um zwei Bytes zu erhalten. Egal, wo man trennte, eine der beiden Teilzahlen wäre immer noch größer als 255, entweder „325“ oder „567“.

Deswegen zerlegt man die Zahl in eine sog. LSB-Zahl („less significant byte“, niederwertiges Byte) und eine MSB-Zahl („most significant byte“, höherwertiges Byte). Die gesamte Zahl setzt sich zusammen aus

Gesamtzahl = 256 x MSB + LSB

Weder MSB noch LSB dürfen dabei größer als 255 sein.

Beispiele dazu:

```
32567 = 256 x 127 +  55, MSB= 127, LSB=  55
32568 = 256 x 127 +  56, MSB= 127, LSB=  56
 2000 = 256 x   7 +208, MSB=   7, LSB= 208
65535 = 256 x 255 +255, MSB= 255, LSB= 255
```

Ein Beispiel soll uns die Vorgehensweise zeigen. Die Zahl, die in zwei Bytes zerlegt werden soll, sei 7800. Dazu teilen wir zunächst 7800 durch 256.

7800 / 256 = 30.47

Damit haben wir bereits das MSB ermittelt. Es ist 30. Nun ergibt aber ...

30 x 256 = 7680

Mithin fehlen noch ...

7800 – 7680 = 120

Das LSB ist also 120 und es ergibt sich:

7800 = 256 x 30 + 120 , MSB = 30, LSB= 120

In Worten: Die Gesamtzahl wird durch 256 geteilt, der Vorkommateil des Ergebnisses stellt das MSB dar. Dieses wird nun mit 256 multipliziert, die Differenz zur Gesamtzahl ist das LSB. Das LSB kann auch 0 sein, nämlich dann, wenn die Gesamtzahl ein Vielfaches von 256 ist.

Das höherwertige Byte („most significant byte", MSB) gibt also an, wie oft die 256 in der Gesamtzahl enthalten ist, das niederwertige Byte („less significant byte", LSB) ergänzt zur vollen Zahl. Statt „MSB" wird auch oft „ADH" („adress higher"), statt „LSB" oft „ADL" („adress lower") gesagt. Dann läßt sich der gezeigte Zusammenhang auch wie folgt ausdrücken:

Adresse = 256 x higher byte + lower byte

Die Abspeicherung erfolgt immer in der folgenden Reihenfolge: erst das LSB, dann das MSB. Es ist wichtig, das zu wissen.

Mathematiker werden erkennen, daß all dies im Grunde einer Rechnung im 256er-Zahlensystem entspricht. So haben wir also neben dem 2er-System (Binärsystem) auch das 256er-System kennengelernt. Da der Computerspeicher grundsätzlich aus Speicherzellen besteht, die nur Zahlen von 0 bis 255 beinhalten können, wird das 256er-System praktisch bei allen Adressvorgängen benutzt. Dies ist allerdings die einzige Gelegenheit, wo bei einem 8-Bit-Prozessor zwei Bytes zu einer logischen Einheit eben einer Adresse verknüpft werden.

Das gilt nicht nur für Adressen, die als Argument auftreten, also zum Beispiel bei ...

5000 STA 40500

..für die Zahl 40500, sondern auch für die Zahl 5000. Schließlich ist diese ja auch eine Adresse, eben die, die uns anzeigt, wo der Programmschritt steht. Innerhalb des Mikoprozessors gibt es einen sog. „Programmzähler" („program counter"), der bei der Abarbeitung eines Assemblerprogramms auf die jeweils aktuelle Adresse zeigt, deren Befehl gerade abgearbeitet wird. Bei der Ausführung des oben gezeigten Programmschritts würde er also auf die Adresse 5000 zeigen. Dieser Programmzähler besteht wiederum aus zwei Bytes, welche die Adresse im 256er-System enthalten. Unsere Beispielzahl 5000 wäre also gespeichert als ...

5000 = 256 x 19 + 136 , MSB= 19, LSB= 136

Beim Programmzähler wird statt MSB und LSB auch oft PCH „program counter high" und PCL („programm counter low") gesagt.

Stellen Sie sicher, daß Sie das bisher Gesagte voll verstanden haben, bevor Sie im Stoff fortfahren. Wir wollen uns nämlich jetzt noch einem weiteren Zahlensystem zuwenden, dem 16er-System, auch Sedezimal- oder Hexadezimalsystem genannt.

Komfortable Assemblersysteme wie ‚T.EX.AS.‘ erlauben es uns, Adressen als eine einzige Zahl einzugeben, die Zerlegung in MSB und LSB wird von ‚T.EX.AS.‘selbständig vorgenommen. Dies gilt ebenso bei der Ausgabe von Zahlen.

Wenn Sie...

5000 STA 7680

..eingeben, wird die Adresse 7680 automatisch in zwei Bytes zerlegt und in den Speicherzellen 5001 (LSB) sowie 5002 (MSB) abgelegt (In 5000 steht der Code für STA). Bei der Ausgabe wird dies wieder rückgängig gemacht, so daß Sie als Programmierer davon praktisch nichts merken.

Es ist aber noch gar nicht sehr lange her, daß Programmierer noch nicht einmal zu träumen wagten von Systemen solcher Komplexität wie ‚T.EX.AS.‘. Auch heutzutage noch bieten die meisten Assembler-Systeme weitaus weniger Komfort.

Nun haben Sie ja selbst gesehen, wie relativ umständlich die Zerlegung einer Adresse in die Zwei-Byte-Form ist. Bei einem einfachen Assembler-System, wo solche Umrechnungen laufend notwendig sind, da das System sie nicht selbständig vornimmt, kann dies zur Qual werden. Um sich das zu ersparen, sind findige Programmierer auf die Möglichkeit gestoßen, im 16er- oder auch Hexadezimalsystem zu arbeiten. Dann sind nämlich Umrechnungen vom und in das Zwei-Byte-Format sehr leicht.

Das Hexadezimalsystem, kurz „Hexsystem“ genannt, ist so weit verbreitet, daß es viele Programmierer gibt, die sich so sehr daran gewöhnt haben, daß sie gar nicht mehr dezimal arbeiten möchten, selbst nicht mit einem so komfortablen Assembler-System wie ‚T.EX.AS.‘. Aus diesem Grund haben wir auch in ‚T.EX.AS.‘ die Möglichkeit eingebaut, einen „Hex-Modus“ zu wählen. Und aus demselben Grund sollten Sie als Assembler-Programmierer auch wissen, wie hexadezimal funktioniert. Wir empfehlen Ihnen jedoch unbedingt, im Dezimalsystem zu programmieren. Schließlich entspricht das Dezimalsystem (10er System) dem üblichen menschlichen Rechnen und es ist bei dem Komfort eines Assembler-Systems wie ‚T.EX.AS.‘ einfach nicht mehr nötig, in einem „maschinenfreundlichen“ Zahlensystem zu arbeiten.

Genug der Vorrede, wenden wir uns der Frage zu, was das Hexadezimalsystem (auch „Sedezimalsystem“ genannt) nun eigentlich ist. Da wir schon das Binärsystem kennengelernt haben, lassen sich einige der dort gezogenen Schlußfolgerungen analog übertragen.

Während es in unserem normalen 10er-System (Dezimalsystem) nur 10 verschiedene Ziffern gibt, nämlich 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9, existieren im 16er-System 16 verschiedene Ziffer-Zeichen. „Ziffer-Zeichen“ statt „Ziffer“ deswegen, weil zum lateinischen Alphabet nun einmal nur 10 verschiedene Ziffern gehören.

Statt für die fehlenden 6 Ziffer-Zeichen nun neue Zeichen zu kreieren, haben sich die schon genannten findigen Programmierer überlegt, einfach die ersten 6 Buchstaben des Alphabets hinzuzunehmen.

Die Ziffern des Hexsystems lauten folglich:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Beispiele für hexadezimale Zahlen sind „20F9“, „A000“, „8C“, „2A“, „23“. Wie das letzte Beispiel „23“ gut zeigt, sind Hexzahlen oftmals nicht von normalen Dezimalzahlen zu unterscheiden. Es hat sich daher eingebürgert, jeder Hexzahl ein Dollarzeichen voranzustellen. Statt „23“ ist es also besser, „$23“ zu schreiben, wodurch die Zahl eindeutig ist.

Genau wie sich wie Sie ja wissen alle Binärzahlen auch dezimal darstellen lassen, so können wir auch Hexzahlen umrechnen.

hexadezimal	dezimal	hexadezimal	dezimal
0	0	30	48
1	1	.	.
.	.	.	.
9	9	3f	63
A	10	40	64
B	11	.	.
C	12	.	.
D	13	4F	79
E	14	50	80
F	15	.	.
10	16	.	.
11	17	5F	95
12	18	.	.
13	19	.	.
14	20	.	.
15	21	AO	160
16	22	.	.
17	23	.	.
18	24	AF	175
19	25	BO	176
1A	26	.	.
1B	27	.	.
1C	28	BF	191
1D	29	.	.
1E	30	.	.
1F	31	.	.
20	32	FO	240
.	.	.	.
.	.	.	.
2F	47	FF	255

Wenn Sie sich die Umwandlungstabelle genauer betrachten, stellen Sie fest, daß die Zahlen eines Bytes von 0 bis 255 in hexadezimal mit nur zwei Stellen darstellbar sind, $00 bis $FF. Und das vereinfacht natürlich die Adressenzerlegung in zwei Bytes erheblich.

Nehmen wir als Beispiel die Zahl $AO5C. Das höherwertige Byte MSB ist $AO, das niederwertige LSB $5C. Das ist wirklich sehr einfach, nicht wahr? Hingegen das zu $AO5C dezimale Äquivalent von 41052 in zwei Bytes zu zerlegen, ist um ein Vielfaches umständlicher.

Betrachten wir ein kurzes Programm als Hexzahlen:

```
$OFAO  LDX   $#41      ..wird gespeichert als..     $A2  $41
$OFA2  STX   $8001                                  $8E  $01  $80
$OFA5  RTS                                          $60
```

Um die Entsprechung „zwei Hexstellen sind gleich einem Byte“ deutlich zu machen, ist es üblich, Hexzahlen grundsätzlich zwei- oder vierstellig zu schreiben. Statt $FAO wählt man also $OFAO. Damit wird klar, $OF ist das MSB und $AO das LSB bei einer Zerlegung. $FFFF ist die höchste Adresse eines 8-Bit-Mikroprozessors, sie entspricht dem Dezimalwert 65535.

Wegen der immer gleichen Stellenzahl und der damit erreichten Formatierung werden Auszüge des Computerspeichers manchal im Hexsystem ausgegeben. Man spricht dann auch von einem sog. „Hexdump“.

```
Beispiel:   1000  A2  41  8E  01  80  E8  8E  02
            1008  80  60  00  02  00  6D  00  60
            1010  22  5B  38  AF  EO  00  50  BC
```

Beachten Sie, daß auch die Adressen hexadezimal ausgegeben werden (also: $1000, $1008, $1010).

So, damit haben wir unserer Pflicht Genüge getan und erklärt, warum manche Programmierer hexadezimal programmieren. Sollte der eine oder andere Punkt dabei offen geblieben sein – umso besser. Denn – wir möchten es noch einmal sagen – mit einem fortgeschrittenen Assembler-System wie 'T.EX.AS.' in hexadezimal zu programmieren, ist nicht nötig. Abgesehen von der Adressenzerlegung – die ein guter Assembler selbst vornimmt – bringt das hexadezimale Zahlensystem nicht viele Vorteile, wohl aber den entscheidenden Nachteil, für uns völlig ungewohnt zu sein. Und selbst erfahrene Hex-Programmierer sind im allgemeinen nicht in der Lage, mit Hexzahlen zu rechnen, also zum Beispiel zwei Hexzahlen zu addieren. Ein Programierer, der zwei Dezimalzahlen nicht addieren kann, dürfte hingegen recht selten anzutreffen sein.

Zusammenfassung

Der Speicherbereich eines Computers wird in „RAM“ und „ROM“ unterteilt. RAM-Speicher kann eingeschrieben und gelesen werden. Außerdem unterscheidet man die verschiedenen Funktionsbereiche: Betriebssystemzwischenspeicher, Programm- und Variablenspeicher, Bildschirmspeicher, Betriebssystem, Sprachinterpreter (meistens BASIC) und Ein-/Ausgabebereich.

Programme und Daten werden in gleicher Weise als Zahlen im Computer gespeichert. Die Unterscheidung erfolgt durch die Zuordnung zu bestimmten Codes, z.B. Assemblerbefehle zu der Assemblercodeliste, Daten zum ASCII-Code.

Zahlen (Adressen) größer als 255 werden im 256er-Zahlensystem in zwei Bytes – MSB und LSB genannt – aufgeteilt. Auf diese Weise bilden zwei Bytes eine logische Einheit. Eine Adresse wird im Speicher immer in der Reihenfolge LSB – MSB abgelegt.

Die Darstellung im Hexadezimalsystem macht die Zwei-Byte-Aufteilung besonders einfach.

Kapitel 6: Das dritte Programm – ADC, SBC, CLC, SEC, CMP, CPX, CPY, BNE.

In diesem Kapitel geht es um zwei sehr wichtige Bereiche der Assemblerprogrammierung, erstens Rechenoperationen und zweitens sogenannte „Bedingte Anweisungen". Wir werden zu beiden Aspekten eine ganze Reihe von neuen Assemblerbefehlen kennenlernen. Aber schließlich sind Sie ja kein völliger Anfänger mehr in Assembler und anhand der Beispielprogramme werden Sie auch die neuen Anweisungen lernen.

Beginnen wir also mit dem Bereich „Rechenoperationen". Das erfordert einiges Umdenken gegenüber der Programmierung in BASIC oder PASCAL, in Assembler gibt es nämlich nur die Operationen plus und minus. Die Befehlsworte dazu sind „ADC" („add with carry", Addieren unter Berücksichtigung des sog. Carry-Flag) und „SBC" („subtract with carry", Subtrahieren unter Berücksichtigung des sog. Carry-Flag). Da das einzige Rechenregister der Akkumulator ist, können wir nur mit Zahlen von 0 bis 255 arbeiten. Dieser stark eingeschränkte Rechenbereich ist auch dafür verantwortlich, daß es ein sog. „Carry Flag" gibt.

Klären wir zunächst kurz, was allgemein unter dem Begriff „Flag" zu verstehen ist. Nun, ein Flag ist nichts anderes als ein Bit. Man kann durchaus auch „Carry-Bit" sagen. Es ist aber üblich, den Ausdruck „Flag" immer dann zu benutzen, wenn ein bestimmtes Bit Entscheidungs- oder Signalcharakter hat.
So kann zum Beispiel ein Bit darüber entscheiden, ob dieser oder jener Programmteil abgearbeitet werden soll. Ein solches Bit könnte man zum Beispiel als „Sprung-Flag" bezeichnen. „Flag" bedeutet übersetzt „Flagge" und wird hier im Sinne von „Signalflagge" benutzt. Manche Programmierer sagen „die Flag", andere „das Flag".

Unser Carry-Flag signalisiert, wenn der Rechenbereich von 0 bis 255 überschritten wird. „Carry" ist dabei mit „Übertrag" zu übersetzen. Wir machen uns das am einfachsten einmal an einem Beispiel klar.

Sehen wir uns dazu die Addition der Zahlen 82 und 55 an:

1. Zahl	82
2. Zahl	+ 55
Übertrag	1
Ergebnis	137

Ohne Berücksichtigung des Übertrages erschiene das völlig unsinnige Ergebnis 37. Der Übertrag zeigt, daß der Rechenbereich von 0 bis 99 überschritten wird.
In Assembler ist das sehr ähnlich, nur daß eben der Rechenbereich von 0 bis 255 reicht. Normalerweise ist das Carry-Flag rückgesetzt, d.h. 0, bei Überschreitung des Bereichs wird es auf 1 gesetzt.

Dabei ist folgendes zu beachten: Eine Addition muß ja, um korrekt zu sein, eventuell schon vorhandene Überträge vorausgegangener Operationen berücksichtigen. Wenn man dies nicht will, so ist es erforderlich, das Carry-Flag vor der Addition rückzusetzen. Dadurch werden falsche Ergebnisse vermieden. Zum Rücksetzen des Carry-Flag dient der Befehl CLC („clear carry“, „lösche Carry“, „lösche Übertrag“).

So, genug der Theorie. So sieht ein Programm in Assembler aus, das die beiden Zahlen 82 und 55 addiert. Die Eingabe wird Ihnen keine Probleme bereiten. Diesmal sind auch keine Adressen dabei, die für unterschiedliche Computersysteme umgestellt werden müßten.

```
5001 LDA #82        ;Akku mit 82 laden
5003 CLC            ;Carry-Flag löschen
5004 ADC #55        ;55 zu Akku-Inhalt addieren
5006 STA 5000       ;Akku in Zelle 5000 speichern
5009 RTS            ;Ende
```

Starten Sie das kurze Programm. Es scheint zunächst nichts zu passieren, wenn Sie sich aber die Speicherzelle 5000 danach ansehen, so enthält sie das Ergebnis 137, also die Summe von 82 und 55. Dieses Nachsehen geschieht in ‚T.EX.AS.‘ sehr einfach mit dem Data-Kommando:

```
DA 5000
```

Von BASIC aus können Sie jede Speicherzelle mit PEEK abfragen.

```
PRINT PEEK (5000)
```

Die Erklärung für unser Programm ist nach den vorausgegangenen Erläuterungen recht einfach. Zunächst wird der Akkumulator mit der Zahl 82 geladen. Rufen Sie sich die Bedeutung des „#-Zeichens bei LDA zurück. Der nachfolgende CLC-Befehl dient dazu, das Carry-Flag zu löschen. Diese Anweisung besteht nur aus dem einen Befehlswort „CLC“ ohne weitere Angaben. Sie ist notwendig, da wir bei unserer Addition keine vorausgegangenen Rechenoperationen berücksichtigen wollen und dürfen, um ein richtiges Ergebnis zu erhalten. Schließlich folgt die „eigentliche“ Additionsanweisung ADC. Die Zahl 55 wird zu dem Akkumulatorinhalt hinzuaddiert. Das Doppelkreuzzeichen „#zeigt auch hier wieder, daß die Zahl 55 gemeint ist, und nicht zum Beispiel die Speicherzelle 55. Das Ergebnis steht nach der Ausführung im Akkumulator. Die STA-Anweisung dient dazu, es in der Speicherzelle 5000 abzuspeichern. Dort kann es dann angesehen oder auch weiterverarbeitet werden. RTS beendet wie üblich unser Assemblerprogramm.

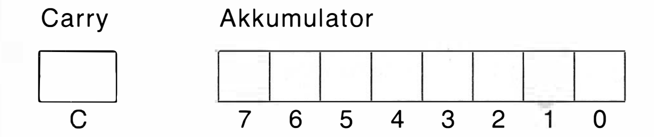
Nebenbei zeigt dieses Programm recht gut die Gleichheit von Programmschritten und Daten. Das Ergebnis der Addition wird in Speicherzelle 5000 transferiert, das Additionsprogramm beginnt direkt danach ab Speicherzelle 5001.

Wir wollen auch für ADC wiederum den Vergleich mit BASIC ziehen.

ADC#arg ..entspricht etwa ... A = A+arg

..wobei ‚arg' ein beliebiges Argument im Bereich 0 bis 255 sein kann. Hier werden die Unzulänglichkeiten eines solchen Vergleichs Assembler/BASIC deutlich: Es gibt in BASIC keine Entsprechung zu unserem Carry-Flag.

Wir wollen uns deswegen die Bedeutung des Carry-Flags noch einmal an einer graphischen Darstellung deutlich machen:



Diese Zeichnung zeigt sehr deutlich die Übertragfunktion des Carry-Bit. Wenn das Ergebnis einer Addition nicht mehr mit den 8 Bits des Akkumulator dargestellt werden kann, so wird quasi als 9. Bit – das Carry-Bit benutzt.

Es gibt Befehle, um abzufragen, ob das Carry-Flag gesetzt – d.h. 1 – ist oder nicht. Wir werden noch darauf eingehen.

Jetzt wollen wir uns erst einmal mit dem Gebiet der mehrfach genauen Addition befassen. Damit bewegen wir uns langsam aber sicher von der Ebene der einfachsten Programmierung auf höhere Ebenen. Stellen Sie daher sicher, daß Sie die vorausgegangenen Kapitel sowie die bisherigen Erklärungen dieses Kapitels wirklich voll verstanden haben. Bei Verständnisschwierigkeiten schlagen Sie einfach nochmal ein paar Seiten zurück.

Es gibt in unserer Assemblersprache nur einen Befehl zur Addition zweier Zahlen im Bereich von 0 bis 255, nämlich ADC. Nun ist es aber möglich – aufbauend auf diesem Befehl – Additionen von theoretisch unbegrenzter Stellenzahl durchzuführen. Wir wollen uns hier auf die Addition von 2-Byte-Zahlen beschränken, da dies in der Praxis häufiger benötigt wird.

Wir haben in Kapitel 5 besprochen, wie zwei Bytes als LSB und MSB zusammengefaßt werden können und somit eine Zahl im Bereich von 0 bis 65535 darstellen.

Indem wir nun mit zwei ADC-Anweisungen beide Bytes einer solchen 2-Byte-Zahl – also LSB und MSB – nacheinander zu den entsprechenden zwei Bytes einer anderen Zahl addieren, können wir tatsächlich in diesem großen Zahlenbereich operieren. Das geht natürlich nur nacheinander, da der Akkumulator in jedem Fall nur ein Byte auf einmal enthalten kann.

Wir wollen für unsere 2-Byte-Addition die folgenden Speicherzellen als Datenspeicher benutzen:

```
5100  LSB1          ;erste Zahl (niederwertiger Teil)
5101  MSB1          ;erste Zahl (höherwertiger Teil)

5102  LSB2          ;zweite Zahl (niederwertiger Teil)
5103  MSB2          ;zweite Zahl (höherwertiger Teil)

5104  LSB3          ;Ergebniszahl (niederwertiger Teil)
5105  MSB3          ;Ergebniszahl (höherwertiger Teil)
```

Diese Festlegung ist willkürlich, sie wird Ihnen aber sehr hilfreich sein beim Verständnis des nun folgenden Programms:

```
5000  LDA  5100         LSB1 in Akku laden
5003  CLC               ;Carry-Flag löschen
5004  ADC  5102         ;LSB2 zu Akku dazuaddieren
5007  STA  5104         ;Ergebnis in LSB3 abspeichern
5010  LDA  5101         ;MSB1 in Akku laden
5013  ADC  5103         ;MSB2 addieren (mit evtl. Carry)
5016  STA  5105         ;Ergebnis in MSB3 abspeichern
5019  RTS               ;Ende
```

Sehen Sie sich das Programm in Ruhe an und überlegen Sie sich die Funktionsweise. Beachten Sie, daß sowohl bei LDA als auch bei ADC das Doppelkreuzzeichen „#“ fehlt. Das bedeutet, das dazugehörige Argument ist eine Adresse und kein direkter Wert. So wird zum Beispiel der Akkumulator nicht mit dem Wert 5100 geladen – was ja auch gar nicht möglich wäre – sondern mit dem Wert, der in Speicherzelle 5100 steht. Dazu addiert wird der Wert, welcher in Speicherzelle 5102 steht usw. Wir werden noch auf diese verschiedenen sogenannten „Addressierungsarten“ zu sprechen kommen.

Probieren wir nun erst einmal unser Programm aus. Dazu müssen wir zunächst die beiden zu addierenden Zahlen eingeben. Dies geschieht in ‚T.EX.AS.‘ auf folgende einfache Weise:

```
5100  820
5102  500
```

Es sollen also die beiden Zahlen 820 und 500 addiert werden. ‚T.EX.AS.‘ erlaubt die Eingabe direkt, die Aufspaltung in LSB und MSB entfällt.

Sie können sich aber jederzeit davon überzeugen, daß die Zerlegung in zwei Bytes korrekt vorgenommen wurde. Geben Sie dazu einfach...

DA 5100 5103

..ein. Neben den Dezimalwerten werden noch andere Daten wie Hex-, ASC- und BSC-Code ausgegeben, die uns aber hier nicht interessieren.

Nachdem wir nun die zu addierenden Zahlen eingegeben haben, starten wir das Programm (EX 5000). Wenn Sie alles richtig gemacht haben, müßte jetzt das Ergebnis 1320 als LSB und MSB in den Speicherzellen 5104/5105 stehen.

Überprüfen Sie das mit dem Double-Data-Kommando:

DD 5104

Es erscheint die Gesamtzahl 1320. LSB und MSB können Sie mittels Data-Kommando ansehen:

DA 5104 5105

Von BASIC aus können Sie das Ergebnis mit der EXBASIC LEVEL II Anweisung...

PRINT DEEK (5104)

..ebenfalls abfragen.

Lassen Sie sich ruhig Zeit beim Austesten unseres Programmes und arbeiten Sie noch mit ein paar weiteren Zahlen. Um die Zahlen x und y zu addieren, geben Sie sie einfach wie folgt ein:

5100 x
5102 y

Sie tippen natürlich nicht „x“ oder „y“ ein, sondern Zahlen zwischen 256 und 65535, wobei die Summe nicht größer als 65535 sein sollte (um falsche Ergebnisse zu vermeiden). Danach starten Sie das Programm (EX 5000) und können sich das Ergebnis mit...

DD 5104

..ansehen. Und wenn Sie das alles verstanden haben, dann dürfen Sie jetzt ein bißchen stolz auf sich sein, denn dann gibt es nichts mehr im Bereich der Assemblerprogrammierung, das Ihnen noch unüberwindbare Schwierigkeiten bereiten wird.

Wir wollen jetzt noch einmal zusammenfassend die wichtigsten Aspekte der 2-Byte-Addition ansprechen und dabei insbesondere die Funktion des Carry-Flag dabei klären.

Um zwei Zahlen zu je zwei Byte zu addieren, wird zuerst das niederwertige Byte der einen Zahl in den Akkumulator geladen. Das Carry-Flag wird gelöscht und das niederwertige Byte der anderen Zahl mit ADC addiert. Das entstehende Zwischenergebnis wird in einer Speicherzelle mittels STA abgelegt. Nun wird das höherwertige Byte der ersten Zahl in den Akkumulator geladen und mit ADC das höherwertige Byte der zweiten Zahl addiert. Das Carry-Flag wird dabei mitaddiert („ADC" heißt ja „add with carry"), je nachdem, ob es von der ersten ADC-Operation gesetzt wurde oder nicht. Auf diese Weise stellt das Carry-Flag den Übertrag vom niederwertigen zum höherwertigen Byte dar. Grundsätzlich ist nach diesem Prinzip eine Arithmetik beliebig hoher Genauigkeit möglich.

So, nachdem wir uns nun ausführlich mit der Addition in Assembler beschäftigt haben, wollen wir uns jetzt der Subtraktion zuwenden. Sie werden feststellen, daß diese im Grunde recht ähnlich dem schon Besprochenen ist, nur eben „umgekehrt".

Das Befehlswort zur Subtraktion lautet „SBC", „subtract with carry", Subtrahieren unter Berücksichtigung des Carry-Flag. Um die Zahl 55 von 82 zu subtrahieren, gehen wir wie folgt vor:

```
5001  LDA  #82        ;Akku mit 82 laden
5003  SEC             ;Carry-Flag setzen
5004  SBC  #55        ;55 von Akku-Inhalt subtrahieren
5006  STA  5000       ;Akku in Zelle 5000 speichern
5009  RTS             ;Ende
```

Neu für Sie ist der Befehl SEC. Nun, während vor einer Addition das Carry-Flag mit CLC („clear carry") gelöscht werden muß, ist es notwendig, vor einer Subtraktion das Carry Flag zu setzen. Und das geschieht mittels SEC („set carry", übersetzt „setze Carry-Flag"). Der Grund dafür: Bei der Addition zweier Zahlen kann es zu einem „Überlauf" des Zahlenbereichs kommen, wenn das Ergebnis im Akkumulator größer als 255 ist. Bei einer Subtraktion kann das sicherlich nicht passieren. Dafür gibt es hier die Gefahr eines „Unterlaufs", nämlich daß das Ergebnis kleiner als 0 ist. Genau wie das Carry-Flag bei der Addition dazu benutzt wird, um ein Ergebnis größer als 255 zu erkennen, dient es bei der Subtraktion zur Handhabung von Ergebnissen kleiner als 0. Da intern im Mikroprozessor die Subtraktion einfach als eine Umkehr der Addition realisiert ist, muß auch das Carry-Flag vor der Subtraktion gesetzt (1) werden – im Gegensatz zur Addition. Und dazu dient eben SEC.

Um unser Programm auszuprobieren, geben Sie in ‚T.EX.AS.' einfach „EX 5001" ein. Das Ergebnis steht danach in Speicherzelle 5000. Sie können es wie bereits erklärt mit dem Data-Kommando ansehen (DA). Es sollte 27 sein.

Bei der Subtraktion gilt bezüglich des Carry-Flag: Es sollte vor SBC mit SEC gesetzt werden. Nach der Ausführung von SBC ist das Carry-Flag unverändert – also immer noch 1 – wenn das Ergebnis der Operation im „erlaubten Zahlenbereich von 0 bis 255 liegt. Ist das Carry-Flag hingegen 0, so bedeutet dies, daß das Ergebnis kleiner als 0 ist („Bereichsunterschreitung"). Sie sollten unbedingt einmal ausprobieren, was im Akkumulator steht, wenn Sie – zum Beispiel – 82 von 55 subtrahieren. Leider kennen wir jetzt im Moment noch keine Möglichkeit, uns anzusehen, ob das Carry-Flag gesetzt ist oder nicht. Wir kommen aber noch dazu.

Unser bisher erworbenes Wissen erlaubt es uns aber bereits jetzt, 2-Byte-Subtraktionen durchzuführen. Für die Datenspeicherung soll dabei die selbe Festlegung wie für die Addition gelten (Seite 6-4). Unter dieser Voraussetzung sieht unser Programm wie folgt aus:

```
5000 LDA  5100        ;LSB1 in Akku laden
5003 SEC              ;Carry-Flag setzen
5004 SBC  5102        ;LSB2 von Akku subtrahieren
5007 STA  5104        ;Ergebnis in LSB3 abspeichern
5010 LDA  5101        ;MSB1 in Akku laden
5013 SBC  5103        ;MSB2 von Akku subtrahieren
5016 STA  5105        ;Ergebnis in MSB3 abspeichern
5019 RTS              ;Programmende
```

Vergleichen Sie das Programm mit dem auf Seite 6-4 abgedruckten für die Addition. Sie werden feststellen, daß nur statt ADC jetzt SBC und statt CLC jetzt SEC steht. Die Struktur hingegen hat sich nicht verändert. Das Carry-Flag dient auch hier wieder als Übertrag, allerdings als „umgekehrter Übertrag". Man sagt dazu auch „borrow" („borgen"), da sich die Subtraktion erforderlichenfalls das Carry-Bit „borgt", um ein korrektes Ergebnis zu erzielen. Dieses Borgen kennen Sie sicher auch vom üblichen Rechnen her.

Hier wird die Zahl 261 von 325 subtrahiert:

1. Zahl	325
2. Zahl	-261
	64

Man rechnet: 5 minus 1 ist gleich 4, 2 minus 6 ist nicht mehr positiv, also wird eine 1 geborgt von der nächsthöheren Stelle, d. h. 12 minus 6 ist gleich 6, die geborgte 1 wird von der 3 abgezogen, was 2 minus 2 und somit 0 ergibt. Wenn bei einem Unterlauf das Carry-Bit geborgt wird, so wird damit praktisch eine Stelle genommen, die im Akkumulator nicht zur Verfügung steht. Das Carry-Flag stellt also in gewisser Weise eine 9. Stelle dar.

Wir wollen uns an dieser Stelle dem zweiten Gebiet des vorliegenden Kapitels zuwenden, den bedingten Anweisungen. Damit ist gemeint, daß bestimmte Anweisungen eines Programms nur abgearbeitet werden, wenn vorher definierte Bedingungen erfüllt sind. Sie kennen so etwas sicherlich von BASIC her.

```
30 IF A=5 THEN 60
```

Diese BASIC-Zeile bewirkt einen Sprung zu Zeilennummer 60, sofern die Bedingung 'A=5' erfüllt ist, d. h., sobald A eben gleich 5 ist. Nun, in Assembler sieht so etwas ein bißchen anders aus.

Lassen Sie uns zunächst ein Programm ausprobieren. Tippen Sie die folgenden Assembleranweisungen wie üblich in 'T.EX.AS.' ein. Beachten Sie dabei, daß diesmal wiederum die erste Bildschirmadresse mit 32768 angegeben ist. Dies muß natürlich wieder an Ihren Computertyp angepaßt werden.

```
5000 LDA      #0        ;Akku mit 0 laden
5002 LDX      #0        ;X-Register mit 0 laden
5004 STA  32768,X       ;Akku in 32768+X-Reg. speichern
5007 CLC                ;Carry-Flag löschen
5008 ADC      #1        ;Akkuinhalt um 1 erhöhen
5010 INX                ;Inhalt des X-Register um 1 erhöhen
5011 CPX      #0        ;X-Reg. mit 0 vergleichen
5013 BNE   5004         ;wenn nicht 0, Sprung zu 5004
5015 RTS                ;anderenfalls (gleich 0) Ende
```

Wenn Sie das Programm eingetippt haben, probieren Sie es mit EX 5000 aus. Beachten Sie, daß bei bestimmten Farbcomputern die Bildschirmfarbe u. U. zuvor umgestellt werden muß, wie in Kapitel 2 besprochen. Andernfalls ist bei diesen Geräten nichts zu sehen. Wenn Sie alles richtig gemacht haben, wird ein Teil des Bildschirms mit dem Zeichenvorrat des Computers gefüllt.

Gehen wir nun das Programm schrittweise durch.

Zunächst wird der Akkumulator mit dem Wert 0 geladen. Beachten Sie das Doppelkreuzzeichen „#“. Als nächstes wird das X-Register ebenfalls mit 0 geladen. Soweit also nichts neues für uns.
Aber jetzt kommt eine neue Variante des STA-Befehls. Sie bedeutet: „Speichere den Inhalt des Akkumulators in der Speicherstelle '32768 plus X-Register' ab“. Wenn das X-Register also eine 0 enthält, so wird damit die Adresse 32768 angesprochen, bei X-Register gleich 1 die Adresse 32769, bei X gleich 2 Adresse 32770 usw. Wir werden gleich sehen, wie sich das auswirkt.
Die beiden folgenden Anweisungen CLC und ADC haben wir eben gerade besprochen, zusammen bewirken sie hier, daß zu dem jeweiligen Inhalt des Akkumulators eine 1 addiert wird. Mit anderen Worten: Der Akkumulator wird um genau eins erhöht.

Der sich anschließende INX-Befehl ist auch nicht neu für Sie. Hier wird der Inhalt des X-Registers um eins erhöht. Mit den Anweisungen CLC, ADC und INX werden also nacheinander sowohl der Akkumulator als auch das X-Register inkrementiert. Sie erinnern sich: Es gibt keinen direkten Befehl zum Inkrementieren des Akkumulators. Deswegen ist auch der „Umweg“ über CLC/ADC nötig.

Jetzt kommt die bedingte Anweisungsfolge. Sie setzt sich zusammen aus der CPX- und der BNE-Anweisung.

Eine bedingte Anweisung besteht grundsätzlich aus zwei Teilen; erstens einem logischen Vergleich und zweitens einem bedingten Sprung. Ziehen wir zur Erklärung noch einmal unsere BASIC-Zeile heran.

```
30 IF A=5 THEN 60
```

Hier besteht der logische Vergleich aus 'A=5', der Sprungbefehl ist 'THEN 60'.

Für unser Assemblerprogramm gilt:

```
5011 CPX  #0         ;logischer Vergleich (X mit 0)
5013 BNE  5004       ;bedingter Sprung zu 5004
```

„CPX“ steht für „compare x-register“, übersetzt „vergleiche X-Register“, in unserem Fall mit 0 ('CPX #0'). Es wird also festgelegt, was womit verglichen werden soll, nämlich der Inhalt des X-Registers mit dem Wert 0. Die CPX-Anweisung enthält aber keinen Hinweis, worauf verglichen wird, also zum Beispiel ob auf gleich, ungleich, kleiner oder größer oder Kombinationen davon (etwa kleiner/gleich).
Dies steht im zweiten Teil der bedingten Anweisungsfolge: 'BNE 5004'. „BNE“ ist die Abkürzung für „branch on not equal“, „verzweige wenn nicht gleich“.

Die CPX- und BNE-Anweisungen zusammengenommen bedeuten also: „Vergleiche das X-Register mit dem Wert 0 und verzweige bei Ungleichheit zu Adresse 5004“. „Verzweige“ ist hier im Sinne von „springe nach“ gemeint. Ist die Bedingung „ungleich 0“ nicht erfüllt, d. h. enthält das X-Register eben eine 0, so wird der Sprung zu Adresse 5004 nicht ausgeführt, sondern der nachfolgende Befehl. In unserem Programm ist das RTS.

Nachdem Sie nun die Bedeutung der einzelnen Assembleranweisungen kennengelernt haben. wollen wir nun betrachten, wie das Programm abläuft, wenn wir es bei Adresse 5000 starten.

LDA und LDX am Programmanfang „initialisieren“ praktisch das Programm. Das bedeutet, die benötigten Register – hier Akkumulator und X-Register – werden mit den gewünschten Werten in diesem Fall 0 belegt. Andernfalls wäre ja nicht sicher, welche Werte diese Register beinhalteten.

Nun folgt von 5004 bis 5013 eine Schleife, hervorgerufen dadurch, daß von 5013 mit BNE unter bestimmten Bedingungen zu 5004 zurückgesprungen wird. Dies entspricht ungefähr der folgenden BASIC-Konstruktion (Zeilennummern):

```
 0 ...
 2 ...
 4 ...
 7 ...
 8 ...
10 ...
11 ...
13 IF .. THEN 4
15 ...
```

„...“ stehen für beliebige gültige Anweisungsfolgen. „..“ ersetzt eine Bedingung, unter der zu Zeile 4 gesprungen wird.

Innerhalb der Schleife wird nun in unserem Assemblerprogramm folgendes getan:

Der Inhalt des Akkumulators wird an eine bestimmte Stelle auf den Bildschirm geschrieben. Diese Stelle ergibt sich aus 32768 (bzw. allgemein Bildschirmanfang) plus dem Inhalt des X-Registers. Während des ersten Schleifendurchlaufs ist das X-Register gleich 0, folglich wird Adresse 32768 angesprochen. Dann werden mit CLC/LDA und INX Akkumulator und X-Register um eins erhöht.
Es folgt der Vergleich des X-Registers mit 0 (CPX). Durch die Erhöhung um eins steht nun auch eine 1 im X-Register, so daß die Sprungbedingung „ungleich 0“ erfüllt ist (BNE). Folglich springt die Programmausführung zu Adresse 5004 zurück. Wiederum wird der Akkumulatorinhalt (jetzt 1) mit STA auf den Bildschirm gebracht, diesmal allerdings an Adresse 32768+1, also 32769, da auch das X-Register jetzt eine 1 enthält.
Erneut werden Akkumulator und X-Register um eins erhöht (CLC/ADC, INX), also auf 2. Auch 2 ist ungleich 0, so daß der Rücksprung zu Adresse 5004 wiederum erfolgt (CPX, BNE).

Auf diese Weise werden nacheinander alle Zeichen des Computers auf dem Bildschirm dargestellt. Da der Akkumulator mit jedem Schleifendurchlauf einen höheren Wert erhält, wird jedesmal ein anderer BSC-Code bzw. das dazugehörige Zeichen auf den Bildschirm gebracht. Der Programmablauf verbleibt in dieser Schleife solange, bis alle 256 verschiedenen Zeichen zu sehen sind (BSC-Codes von 0 bis 255). Dann ist die Bedingung „X-Register ungleich 0“ nicht mehr erfüllt und das Programm fährt bei RTS fort.

Um das verstehen zu können, müssen Sie sich in Ihr Gedächtnis zurückrufen, daß sowohl Akkumulator als auch X-Register nur Zahlen von 0 bis 255 beinhalten können. Wird dieser Bereich nach oben überschritten, so wird wieder von vorne angefangen zu zählen. Mit anderen Worten: Auf 255 folgen 0, 1, 2, 3 usw. Und zu diesem Effekt kommt es auch in unserem Programm, wo mit CLC/ADC und INX immer eins addiert wird. Es wird von 0 bis 255 gezählt, bei dem Versuch, 256 einzuschreiben, beginnt dies wieder von 0.

In dem Moment, wo das X-Register wieder eine 0 enthält, ist die Programmschleife beendet und es wird RTS ausgeführt.

Zur Verdeutlichung wollen wir einmal unserem Assemblerprogramm ein entsprechendes in BASIC gegenüberstellen.

```
5000 LDA       #0          0  A=0
5002 LDX       #0          2  X=0
5004 STA  32768,X          4  POKE 32768+X,A
5007 CLC
5008 ADC       #1          8  A=A+1
5010 INX                  10  X=X+1
5011 CPX       #0
5013 BNE   5004           13  IF  X()256 THEN 4
5015 RTS                  15  END
```

Die Variable A steht anstelle des Akkumulators, die Variable X für das X-Register. Zu CLC existiert in BASIC keine Entsprechung, die Funktion ist jedoch in der BASIC-Zeile 8 enthalten. Ebenso sind die Funktionen von CPX und BNE in einer BASIC-Zeile (13) enthalten. Da eine Variable in BASIC nicht bei 256 wieder von 0 anfängt, ist es nötig, 256 statt 0 zu schreiben.

„Spielen" Sie ruhig noch ein bißchen mit dem Assemblerprogramm, um es voll zu verstehen. Ändern Sie doch zum Beispiel einmal folgendes:

```
5011 CPX #100
```

Starten Sie mit EX 5000. Was passiert?

Andere Veränderungen können sein ...

```
5000 LDA    #50

5002 LDX    #50

5004 STA 32850,X

5008 ADC      #2
```

Die Abstände zwischen den Zeilen sollen andeuten, daß diese Modifikationen nicht alle auf einmal vorgenommen werden müssen. Sie sollten sie aber alle nacheinander ausprobieren und können sie auch beliebig mischen. Sie können das Programm natürlich auch noch viel weitergehender als hier gezeigt verändern. Denken Sie aber auch daran, daß Sie bei Programmfehlern u. U. den Computer kurz ausschalten müssen, um ihn zu reaktivieren.

Der Vollständigkeit halber wollen wir jetzt noch zwei weitere Assemblerbefehle ansprechen. Es sind CPY und CMP. Beide sind dem uns bereits bekannten Befehl CPX ähnlich, CPY bezieht sich aber auf das Y-Register, CMP auf den Akkumulator. Ersetzen Sie in unserem Programm einmal die CPX-Anweisung wie folgt:

```
5011 CMP #0
```

Statt des X-Registers wird hier also der Akkumulator mit 0 verglichen. Der Austausch ist in unserem Programm erlaubt, da der Akkumulator ebenso wie das X-Register bei jedem Schleifendurchlauf um jeweils eins erhöht wird. Dies gilt jedoch nicht allgemein und es darf zum Beispiel nicht das Y- statt des X-Registers gewählt werden.

„CPY" steht für „compare y-register", „vergleiche Y-Register". „CMP" bedeutet „compare akkumulator", „vergleiche Akkumulator". Die Zahl, mit der verglichen werden soll, folgt dem Befehlswort mit Doppelkreuz. Beispiele dafür:

```
CMP  #180          ;Akkumulator
CPX  #255          ;X-Reg., größtmögliche Zahl (255)
CPY  #0            ;Y-Reg., kleinstmögliche Zahl (0)
CPY  #255
CMP  #300          ;FALSCH, größer als 255
```

Beachten Sie, daß keine dieser Vergleichsanweisungen bereits eine Relation herstellt. So wird mit 'CMP #180' zwar der Akkumulatorinhalt mit der Zahl 180 verglichen, jedoch noch nicht festgelegt, ob auf gleich, ungleich, kleiner oder größer usw. Deswegen existiert dazu keine Entsprechung in BASIC.
Vielmehr folgt praktisch jeder Vergleichsanweisung ein bedingter Sprung, in unserem Beispielprogramm war dies BNE. Erst beide Anweisungen zusammengenommen haben eine BASIC-Entsprechung.

Es gibt in Assembler eine ganze Reihe von bedingten Anweisungen. Wir werden dies im nächsten Kapitel gründlich durcharbeiten und dabei gleichzeitig die dahintersteckende theoretische Konzeption des Flag-Registers besprechen. Wir wollen Ihnen aber schon vorher an einem praktischen Beispiel das Gefühl für dieses Thema vermitteln und hoffen, daß uns das gelungen ist. Schließlich erfordert eine Programmiersprache wie jede andere Sprache auch neben der Kenntnis der Grammatik (Syntax) Erfahrung im Umgang damit. Für Assembler gilt das in besonders hohem Maße. Das ist der Grund dafür, daß Sie sich jetzt obgleich schon bei Kapitel 6 angelangt vermutlich noch nicht in Assembler sicher fühlen. Das sollte aber kein Grund zur Sorge sein, schließlich sind Sie noch Lernender.

Zusammenfassung

Es existieren in Assembler zwei Befehle zum Addieren und Subtrahieren von 1-Byte-Zahlen:

```
ADC #argument
SBC #argument
```

ADC addiert das Argument zum Akkumulatorinhalt, SBC subtrahiert das Argument vom Akkumulatorinhalt. Dies geschieht unter Berücksichtigung des Carry-Flags als Übertrag (Über- oder Unterlauf). Vor einer ADC-Anweisung muß das Carry-Flag grundsätzlich mit CLC gelöscht, vor einer SBC-Anweisung mit SEC gesetzt werden. Ist das Ergebnis der Addition größer als 255, so wird Carry gesetzt, ist das Ergebnis der Subtraktion kleiner als 0, so wird Carry gelöscht.

Wird das Carry-Flag vor der Operation nicht gelöscht bzw. gesetzt, so wird es als Übertrag berücksichtigt. Auf diese Weise ist eine Arithmetik beliebiger Genauigkeit möglich.

Mit der sog. „indizierten Adressierung“ kann die Adresse zu STA vom Inhalt des X-Registers abhängig gemacht werden.

```
STA adresse,X
```

Der Inhalt des Akkumulators wird in diejenige Speicherzelle gebracht, deren Adresse sich aus 'adresse plus Inhalt des X-Registers' errechnet.

Eine bedingte Anweisung besteht in Assembler im allgemeinen aus einer Vergleichsanweisung und einem bedingten Sprungbefehl. Vergleichsanweisungen sind:

```
CMP #argument
CPX #argument
CPY #argument
```

CMP vergleicht das Argument (Bereich von 0 bis 255) mit dem Inhalt des Akkumulators. CPX wirkt entsprechend für das X-Register, CPY für das Y-Register.

```
BNE sprungadresse
```

BNE ist ein bedingter Sprungbefehl, der auf eine Vergleichsanweisung folgen kann. Die Programmausführung wird an der Sprungadresse fortgesetzt, sofern das Ergebnis der vorausgegangenen Operation ungleich 0 ist. Andernfalls fährt das Programm mit dem direkt auf BNE folgenden Befehl fort.

Kapitel 7: Adressierungsarten und Vergleiche – BCC, BCS, BEQ, BMI, BPL, BVC, BVS, JMP.

Im vorangegangenen praxisorientierten Kapitel haben wir die Themen Adressierungsarten und Vergleichsanweisungen bereits angesprochen. Beide sind jedoch so grundlegender Natur, daß wir nicht umhinkommen, sie detailiert zu erklären. Beginnen wir mit dem Begriff „Adressierungsart".

Unter „Adressierungsart" verstehen wir die Art und Weise, in welcher die Argumentadresse einer Assembleranweisung wirkt. Nehmen wir als Beispiel die beiden folgenden STA-Befehle – beide sind Ihnen nicht neu:

```
STA 32768        ;speichert Akku nach 32768
STA 32768,X      ;speichert Akku nach 32768 + X-Register
```

Das Befehlswort ist in beiden Fällen STA, die unterschiedliche Adressierungsart bewirkt jedoch auch unterschiedliche Bedeutungen der Anweisungen. Im ersten Fall spricht man von „absoluter Adressierung", im zweiten von der „X-indizierten Adressierung". Es existieren insgesamt sieben verschiedene Adressierungsarten des STA-Befehls, andere Befehlsworte haben weniger oder noch mehr Adressierungsarten. Einige Assemblerbefehle kennen nur eine einzige Adressierungsart.

Anhand des uns ebenfalls schon bekannten Befehls LDA wollen wir uns acht Adressierungsarten ansehen:

```
LDA #argument                  ;unmittelbar
LDA zeropage-adresse           ;Zero-Page
LDA zeropage-adresse, x        ;Zero-Page indiziert
LDA adresse                    ;absolut
LDA adresse, x                 ;indiziert
LDA adresse, y                 ;indiziert
LDA (zeropage-adresse, x)      ;indiziert-indirekt
LDA (zeropage-adresse), y      ;indirekt-indiziert
```

Das ist auf den ersten Blick recht verwirrend, wir werden aber jetzt nacheinander alle acht Adressierungsarten besprechen. Lesen Sie sich die verschiedenen Möglichkeiten in Ruhe durch. Wichtig ist zunächst nicht, daß Sie gleich alles behalten, sondern daß Sie alles verstehen. Bei Bedarf können Sie später jederzeit nachschlagen.
Beachten Sie, daß die Erklärungen nur exemplarisch am LDA-Befehl geschehen, aber die Adressierungsarten als solche existieren für unterschiedliche Assemblerbefehle. Allerdings weist nicht jeder Befehl alle Adressierungsarten auf. Sie sehen ja auch schon an obiger Aufstellung, daß bei LDA bestimmte Adressierungsmöglichkeiten nur für das X- oder das Y-Register, jedoch nicht für beide, existieren.

Unmittelbar (immediate) – LDA #argument

Dem Befehlswort folgt unmittelbar das dazugehörige Byte-Argument. Unter „Byte-Argument“ verstehen wir dabei eine Zahl, die innerhalb eines Bytes dargestellt werden kann. Sie muß also im Bereich von 0 bis 255 liegen.

Das entsprechende Register – bei LDA der Akkumulator – wird direkt mit dem Wert geladen oder auch zum Beispiel damit verglichen (CMP). Bei dem Argument handelt es sich also gar nicht um eine Adresse, dennoch sagt man auch hierzu „Adressierungsart“.

Zu erkennen ist die unmittelbare Adressierung immer an dem Doppelkreuzzeichen. Die folgenden Assemblerbefehle können sie verwenden:

LDA, LDX, LDY, ADC, SBC, CMP, CPX, CPY, AND, ORA, EOR.

Sie kennen bereits alle Befehle bis auf die letzten drei. Denken Sie bei jedem Befehlswort einmal kurz darüber nach, was es bewirkt.

Beispiele:

```
LDA # 50        ;lädt Akku mit dem Wert 50
LDA #300        ;FALSCH, da größer als 255
CPX #255        ;vergleicht X-Register mit 255
STA #100        ;FALSCH, für STA nicht zulässig
```

Zero-Page (zero page) – LDA zeropage-adresse

Um das zu verstehen, müssen wir zunächst den Begriff „Zero Page“ klären, „Zero Page“ bedeutet übersetzt soviel wie „Null-Seite“ und meint den Adressbereich von 0 bis 255.

Wie Sie wissen, werden Adressen üblicherweise in zwei Bytes als LSB und MSB abgespeichert. Für den Adressbereich von 0 bis 255 ist dies aber unnötig, da ein Byte zur Darstellung ausreicht. Das MSB wäre nämlich immer 0, so daß es auch weggelassen werden könnte. Die Tatsache, daß das MSB im Adressbereich von 0 bis 255 durchweg 0 ist, hat auch zu dem Begriff „Zero Page“ („Null-Seite“) geführt. Man spricht auch von Page 1, Page 2 usw. nach dem Wert des MSB.

Die Page 0 nimmt eine besondere Stellung ein, da es für sie eine spezielle Adressierungsart gibt eben die Zeropage-Adressierung. Das Besondere daran ist, daß die Adresse in nur einem Byte gespeichert wird. Das ist platzsparend und schneller in der Ausführungszeit des jeweiligen Befehls.

Zeropage-Adressierung gibt es für folgende Befehle:

LDA, LDX, LDY, STA, STX, STY, DEC, INC, CMP, CPX, CPY, ADC, SBC, AND, ORA, EOR, BIT, ROR, ROL, ASL, LSR.

Beispiele:

```
LDY 255        ;lädt Y-Reg. mit Inhalt von Adresse 255
LDA  50        ;lädt Akku mit Inhalt von Adresse 50
DEC   0        ;dekrementiert Adresse 0
STA 300        ;KEINE Zeropage-Adressierung
```

Die Vorteile der Zeropage-Adressierung (Adressbereich 0 bis 255) wird an folgendem Beispiel besonders deutlich:

```
5000 STA 255        5000 STA 300
5002 RTS            5003 RTS
```

Im links gezeigten Beispiel reichen zwei Speicherzellen zur Speicherung der Anweisung 'STA 255' aus, der nächste Befehl folgt bei Adresse 5002. Im rechten Beispiel ist Zeropage-Adressierung nicht möglich, die Adresse 300 paßt nicht mehr in ein einziges Byte, folglich benötigt die gesamte Anweisung drei Speicherzellen (5000, 5001 und 5002).

Ein einheitliches besonderes Zeichen zur Darstellung der Zeropage-Adressierung – etwa ähnlich dem Doppelkreuz bei unmittelbarer Adressierung – existiert nicht. 'T.EX.AS.' nimmt selbständig Zeropage an, wenn die Adresse im zulässigen Bereich liegt und das Befehlswort das zuläßt.

Zero-Page indiziert (zero page indexed) – LDA zeropage-adresse, X
LDX zeropage-adresse, Y

Die Adressierungsart „indiziert" ist uns schon im letzten Kapitel bei STA begegnet. Zu der angegebenen Adresse wird der Inhalt des X- bzw. Y-Registers addiert und die daraus resultierende neue Adresse wird berücksichtigt. Das entsprechende Register wird auch als „Index-Register" bezeichnet.

Beispiel:

```
INC 50,X              ;Inhalt des X-Reg. sei z.B. 32
```

Inkrementiert wird bei diesem Beispiel Speicherzelle 82. Adresse 50 selbst bleibt hingegen unverändert.

„Zero Page X-indiziert" gibt es für folgende Befehlsworte:

LDA, LDY, STA, STY, ADC, SBC, DEC, INC, CMP, AND, ORA, EOR, ASL, LSR, ROL, ROR.

„Zero Page Y-indiziert" existiert für: LDX, STX.

Achtung: Bei Überschreitung des Zero-Page-Bereiches von 0 bis 255 durch Addition des Indexregisters wird wieder von vorne an gezählt, es kann also in keinem Fall eine Adresse größer als 255 angesprochen werden.

Beispiele:

```
LDA    55,X     ;lädt Akku mit Inhalt von 55 + X
CMP     1, X    ;vergleicht Akku mit Inhalt von 1 + X
STA   300, X    ;KEINE Zeropage-Adressierung
STA   100, Y    ;KEINE Zeropage-Adressierung
STX   100, Y    ;speichert X-Reg. nach 100 + Y
ADC    50, X    ;addiert Inhalt von 50 + X zum Akku
STX    50, X    ;KEINE Zeropage-Adressierung
```

Machen Sie sich jeweils klar, warum es sich bei einzelnen Beispielen um keine Zeropage-Adressierung handelt.

Auch bei Zeropage-indizierter-Adressierung paßt der gesamte Befehl in zwei Bytes. Im ersten steht das Befehlswort, im zweiten die Adresse im Bereich von 0 bis 255. Beachten Sie, daß bei einer Anweisung wie zum Beispiel ...

```
6000 LDA  100, X
```

.. das „X“ nicht gesondert abgespeichert wird. In diesem Beispiel steht in Speicherzelle 6000 die Zahl 181, in 6001 die Zahl 100. Die Adressierungsart wird bestimmt durch den Assemblercode. Für jedes Assemblerbefehlswort existieren so viele unterschiedliche Codes wie es Adressierungsarten für das Befehlswort gibt.

So wird zum Beispiel LDA je nach Adressierungsart wie folgt abgespeichert:

LDA-Adressierungsart	Assemblercode
unmittelbar	169
Zero Page	165
Zero Page X-indiziert	181
absolut	173
absolut X-indiziert	189
absolut Y-indiziert	185
Zero Page indiziert-indirekt	161
Zero Page indirekt-indiziert	177

Sie sehen, auch zum Beispiel das Doppelkreuzzeichen für „unmittelbar“ wird nicht abgespeichert, vielmehr wird ein entsprechender Assemblercode gewählt, der eben genau diese Adressierungsart meint.

Die Zeichen „#“, „X“, „Y“, Komma usw. dienen lediglich zum Eingeben und Lesen eines Programmes, die Zuordnung zu den entsprechenden Assemblercodes erledigt jedes gute Assemblersystem selbständig.

Absolut (absolute) LDA adresse

Als „absolut“ bezeichnet man eine Adressierungsmöglichkeit, die wir als eine der ersten kennengelernt haben. Dem Befehlswort folgt eine 2-Byte-Adresse im Bereich von 0 bis 65535. Bei dem Beispiel mit LDA wird der Akkumulator mit dem Inhalt der durch die Adresse spezifizierten Speicherzelle geladen.

Beispiele:

```
LDA 50000        ;lädt Akku mit dem Inhalt von 50000
STA 50000        ;speichert Akkuinhalt nach 50000
JMP  4000        ;springt zu Adresse 4000 („jump“)
```

Die Adressierungsart „absolut“ ist für viele Assemblerbefehle vorhanden. Im einzelnen sind das:

LDA, LDX, LDY, STA, STX, STY, DEC, INC, CMP, CPX, CPY, ADC, SBC, JMP, JSR, AND, ORA, EOR, BIT, ROR, ROL, ASL, LSR.

Anweisungen mit absoluter Adressierung sind immer 3-Byte-Befehle, das erste Byte belegt das Befehlswort, die zwei anschließenden die Adresse als LSB und MSB.

Absolut indiziert (absolute indexed) - LDA adresse, X
LDA adresse, Y

Dies entspricht weitgehend der Adressierungsart „Zero-Page indiziert“, die wir bereits besprochen haben. Hauptunterschied: die Adresse ist keine Zeropage-Adresse, sie liegt also nicht notwendigerweise im Bereich von 0 bis 255.

Absolut indizierte Adressierung kann sowohl mit dem X- als auch mit Y-Register vorgenommen werden.

Beispiele:

```
LDA 2500, X      ;lädt Akku mit Inhalt von 2500 + X
LDA 2500, Y      ;lädt Akku mit Inhalt von 2500 + Y
CMP 6000, X      ;vergleicht Akku mit 6000 + X
```

Die folgend aufgeführten Befehle können absolut X-indiziert werden:

LDA, LDY, STA, DEC, INC, CMP, ADC, SBC, AND, ORA, EOR, ROR, ROL, ASL, LSR.

Zur absolut Y-indizierten Adressierung stehen die folgenden Befehle zur Verfügung:

LDA, LDX, STA, DEC, INC, CMP, ADC, SBC, AND, ORA, EOR.

Page-Überschreitung ist – im Gegensatz zu „Zero Page indiziert“ – möglich.

Indiziert-indirekt (indexed indirect) - LDA (zeropage-adresse, X)

Jetzt wird es ein bißchen komplizierter, wir kommen zu der indirekten Adressierung.

Machen wir uns nochmals den Begriff „indiziert klar. Der entsprechende Befehl greift nicht auf die angegebene Adresse zu, sondern errechnet sich aus dieser und dem Inhalt des X- oder Y-Registers eine neue Adresse. Diese wird schließlich angesprochen (LDA, STA etc.).

„Indirekte Adressierung“ meint nun, es wird nicht der Inhalt der angegebenen oder errechneten Adresse zum Beispiel mit LDA geladen, sondern diese Adresse wird als Verweis auf eine weitere Adresse gewertet, deren Inhalt schließlich in den Akkumulator geladen wird.

Ein Beispiel zeigt Ihnen auch sofort, wie das gemeint ist. Beachten Sie: Das Ergebnis von Adresse plus X-Register muß im Zeropage-Bereich liegen.

```
5000 LDA   (61,X)
```

Nehmen wir außerdem an:	Inhalt des X-Registers	:	10
	Inhalt von Adresse 71	:	200
	Inhalt von Adresse 72	:	4

Nun passiert folgendes: Der Inhalt des X-Registers (10) wird zu der angegebenen Adresse (61) addiert, was 71 ergibt. Damit ist aber noch nicht die endgültige Adresse gefunden. Vielmehr interpretiert der Mikroprozessor den Inhalt von 71 als LSB einer 2-Byte-Zahl, deren MSB folglich in 72 stehen muß. In unserem Beispiel ergibt sich daraus die Adresse 4 x 256 + 200 ist gleich 1224. der Akkumulator wird also mit dem Inhalt von Speicherzelle 1224 geladen. Kompliziert? Vielleicht, aber eigentlich auch recht logisch.

Man kann es nämlich auch so betrachten: Eine Klammer um eine Adresse bedeutet immer, daß nicht der Inhalt dieser Adresse gefragt ist, sondern daß diese Adresse zusammen mit der folgenden (LSB und MSB) einen Verweis auf eine andere Adresse darstellt, deren Inhalt geladen oder verändert werden soll; deswegen „indirekt“. Da in unserem Fall das X innerhalb der Klammer steht, muß es zu der angegebenen Adresse addiert werden, bevor indirekt interpretiert wird. Wir werden sehen, daß dies bei Verwendung des Y-Registers genau umgekehrt ist.

Bis auf eine Ausnahme ist die indirekte Adressierung nur zusammen mit einer Indizierung verwendbar. Deswegen ist sie auch etwas schwerer zu verstehen. Es ist aber wichtig, daß Sie die indiziert-indirekte Adressierung verstanden haben. Deswegen sollten Sie jetzt ruhig noch einmal die vorausgegangenen Zeilen lesen.

Ein Schaubild verdeutlicht Ihnen die indiziert-indirekte Adressierung:

```
LDA  (adresse,X)
                      X-Reg.
     adresse + 0    – – LSB1 – –
                                  – – – – Zugriffsadresse 1
     adresse + 1    – – MSB1 – –
     adresse + 2    – – LSB2 – –
                                  – – – – Zugriffsadresse 2
     adresse + 3    – – MSB2 – –
        .
        .
     adresse+255
```

Was Sie hier sehen, ist eine der häufigsten Anwendungen der indiziert-indirekten Adressierung. Aus einer Tabelle oder Liste von Adressen wird mit Hilfe des X-Indexes der Verweis auf eine andere Adressenliste (hier „Zugriffsadresse") gewonnen.

Diese Assemblerbefehle können indiziert-indirekt adressiert werden:

LDA, STA, CMP, ADC, SBC, AND, ORA, EOR.

Indirekt-indiziert (indirect indexed) - LDA (zeropage-adresse), Y

Während die Adressierungsmöglichkeit „indiziert-indirekt" ausschließlich für das X-Register existiert, gilt das für „indirekt-indiziert" ausschließlich für das Y-Register. Ein Schaubild soll Ihnen die Unterschiede klarmachen.

```
LDA  (adresse,Y)
                                         Y-Reg.
     adresse        – – LSB – –
                                  – – – – Zugriffsadresse + 0
     adresse + 1    – – MSB – –
                                  – – – – Zugriffsadresse + 1
                                  – – – – Zugriffsadresse + 2
                                                 .
                                                 .
                                          Zugriffsadresse + 255
```

Die indirekt-indizierte Adressierung ist ausschließlich im Zeropage-Bereich, also bei Adressen von 0 bis 255, möglich.

Bei der indirekt-indizierten Adressierung wird also erst die angegebene Adresse indirekt interpretiert und von der so errechneten Zugriffsadresse das Y-Register als Index ausgewertet. Beachten Sie die Unterschiede der Klammersetzung bei indiziert-indirekter und indirekt-indizierter Adressierung. Wenn Sie die Klammer als Symbol für indirekt ansehen, wird die Schreibweise äußerst logisch.

Beispiel zu indirekt-indizierter Adressierung:

```
5000 LDA (61), Y
```

Nehmen wir außerdem an:	Inhalt des Y-Registers	: 10
	Inhalt von Adresse 61	: 200
	Inhalt von Adresse 62	: 4

Was nun passiert ist folgendes: Die Adressenangabe 61 wird indirekt interpretiert, d. h., 61 und 62 werden als LSB und MSB einer Verweisadresse berechnet. In unserem Fall ergibt sich daraus Adresse 4 × 256 + 200 ist gleich 1224. Zu dieser Adresse wird nun als Index der Inhalt des Y-Registers addiert. Das bewirkt letztlich, daß der Akkumulator mit dem Inhalt der Speicherzelle mit der Adresse 1234 geladen wird. Vergleichen Sie das Beispiel mit dem für die indiziert-indirekte Adressierung.

Die indirekt-indizierte Adressierung ist bei den folgenden Befehlen möglich, wie gesagt, ausschließlich mit dem Y-Register:

LDA, STA, CMP, ADC, SBC, AND, ORA, EOR.

Implizit (implied) CLC

Die Adressierungsart „implizit“ ist zwar nicht im Zusammenhang mit der Anweisung LDA möglich. Wir haben diese Adressierungsart aber schon bei so vielen anderen Befehlen benutzt, ohne besonders darauf hinzuweisen, daß wir sie nun hier auch formal besprechen wollen.

Implizite Adressierung liegt bei jedem 1-Byte-Befehl vor, d.h., wenn kein Argument angegeben ist. In diesem Fall drückt das Befehlswort selbst bereits aus, welches Register gemeint ist.

Beispiele:

```
INX     ;inkrementiert X-Register
DEY     ;dekrementiert Y-Register
CLC     ;löscht Carry-Flag
SEC     ;setzt Carry-Flag
DEX     ;dekrementiert X-Register
INY     ;inkrementiert Y-Register
RTS     ;beendet Programm/Unterprogramm
```

Implizit adressieren die folgenden 1-Byte Befehle:

DEX, DEY, INX, INY, NOP, TAX, TAY, TXA, TYA, TSX, TXS, PHA, PHP, PLA, PLP, RTI, RTS, SEC, CLC, SED, CLD, SEI, CLI, CLV, BRK.

Einige davon haben wir ja bereits kennengelernt. Die restlichen werden wir noch besprechen.

Akkumulator (Accumulator) ASL

Der Vollständigkeit halber wollen wir die Adressierungsart „Akkumulator“ an dieser Stelle erklären. „Akkumulator“ adressieren alle Assemblerbefehle, welche direkt und ausschließlich auf den Akkumulator Bezug nehmen. Es sind die nachfolgenden aufgeführten 1-Byte-Befehle.

ASL, LSR, ROL, ROR.

Es handelt sich hier durchweg um Verschiebe- und Rotations-Anweisungen. Wir werden noch darauf zu sprechen kommen.

Absolut indirekt (absolute indirect) – JMP (adresse)

Die Adressierungsart „absolut indirekt“ existiert nur für einen einzigen Befehl, nämlich JMP. „JMP“ steht für „jump“, übersetzt „Sprung“, und bedeutet „springe nach Adresse...“ Wir wollen die Gelegenheit nutzen und gleich den JMP-Befehl mit kennenlernen.

Beispiele für JMP (absolute Adressierung):

```
JMP  6000      ;setzt Programmausführung bei 6000 fort
JMP 30125      ;setzt Programmausführung bei 30125 fort
```

Der Vergleich mit BASIC ist in diesem Fall recht einfach:

```
JMP adresse  ... entspricht etwa ..  GOTO zeilennummer
```

Die Gleichsetzung von „Adresse“ und „Zeilennummer“ ist zwar nicht ganz korrekt, Sie sind aber mittlerweile versiert genug in Assembler, um sich dadurch nicht verwirren zu lassen. Für JMP ist die Gleichsetzung durchaus sinnvoll.

Kommen wir nun zu der absolut indirekten Adressierung. Sie ist, wie bereits gesagt, nur bei JMP zulässig. „Indirekt“ bedeutet, daß die angegebene Adresse nicht direkt gemeint ist, sondern daß die Adresse vielmehr zusammen mit der nachfolgenden als LSB und MSB einen Verweis auf eine andere Adresse darstellt. Im Zusammenhang mit indizierten Adressierungen haben wir das ja schon kennengelernt.

Allgemein sieht das so aus:

JMP (adresse)

adresse	– – LSB – –	– – – – Sprungadresse
adresse + 1	– – MSB – –	

Beispiel:

JMP (6000)

Die Speicherzellen 6000 und 6001 werden als LSB und MSB interpretiert und es erfolgt ein Sprung zu der sich daraus ergebenden Adresse.

Mit „JMP“ indirekt kann sich ein Programm auf sehr einfache Weise eine Sprungadresse selbst errechnen. Das entspräche im Grunde einer Anweisung „GOTO X“ in BASIC, wobei X eine beliebige Variable sein kann.

So, damit haben Sie alle überhaupt möglichen Adressierungsmöglichkeiten kennengelernt bis auf eine, die „relative Adressierung“. Da diese aber ganz stark mit dem Thema Bedingte Verzweigungen verknüpft ist, wollen wir sie erst einige Seiten später im dazugehörigen Rahmen besprechen.

Es ist sicher sinnvoll, wenn Sie jetzt erst einmal eine Pause einlegen. Wir werden nämlich jetzt anschließend das Thema Bedingte Verzweigungen ebenso komplex wie zuvor Adressierungsarten abhandeln. Daß es dabei ähnlich kompliziert zugehen wird, dürfen wir Ihnen jetzt schon versprechen. Also, legen Sie lieber das Buch für eine Weile aus der Hand oder wiederholen Sie die letzten Seiten, bevor Sie weitermachen.

Damit Sie voll verstehen können, wie bedingte Anweisungen in Assembler funktionieren, ist es notwendig, daß Sie zunächst das sog. „Flag- oder Statusregister“ kennenlernen. Das Statusregister ist ein Teil des Mikroprozessors und gibt den Status oder Zustand des Prozessors bei den einzelnen Assembleranweisungen wieder.

Das Statusregister besteht aus einem Byte, dessen einzelne acht Bits verschiedene, genau bestimmte Flag-Funktionen haben. Wir haben bereits das Carry-Flag, das ein Bit dieses Registers ist, besprochen.

Anstelle des Ausdrucks „Statusregister“ wird häufig auch „P-Register“ gesagt, nämlich als Abkürzung für „Processor Status Register“. Sehen wir uns an, was es damit auf sich hat.

Die 8 Bits des Statusregisters mit ihren Bedeutungen:

N	V	–	B	D	I	Z	C

N: begativ result (negatives Ergebnis)
V: overflow (Überlauf)
–: Erweiterungsbit für zukünftige Verwendung
B: break command (Break-Befehl)
D: decimal mode (Dezimalbetrieb)
I: interrupt disable (Interrupt sperren)
Z: zero result (Null-Ergebnis)
C: carry

Gehen wir nun die Bedeutungen nacheinander durch. Dabei werden wir einige Statusbits ausführlich durchsprechen, bei anderen hingegen die Funktionen nur andeuten, je nachdem, von welcher Wichtigkeit sie für unser Thema Bedingte Anweisungen sind.

N – Negatives Ergebnis (negativ result)

Wie Sie wissen, können die Prozessor-Register wie Akkumulator, X- und Y-Register nur positive Zahlen von 0 bis 255 beinhalten. Das gilt auch für alle Speicherzellen.
Da aber manchmal das Operieren auch mit negativen Zahlen unerläßlich ist, hat man sich einen einfachen aber wirksamen Trick ausgedacht. Statt der acht Bits eines Bytes werden nur sieben zur Darstellung der Zahl benutzt, das achte dient als sog. Vorzeichenbit. Damit reicht der Zahlenbereich von -128 bis +127.

Beispiel:

Bit 7 6 5 4 3 2 1 0
      1 1 0 1 0 1 1 0 = 214 = -42

Vorzeichenbit: 0 = positiv oder Null
               1 = negativ

Sie sehen, je nachdem, ob das achte Bit zur Zahl zugehörig oder als Vorzeichen interpretiert wird, ergibt sich eine andere Zahl.

Das N-Bit (N-Flag) ist immer gleich dem achten Bit des Bytes, egal, ob dieses nun als Vorzeichen gelesen wird oder nicht.

Sie wissen bereits, daß das Carry-Flag unter bestimmten Bedingungen bei arithmetischen Operationen (ADC und SBC) vom Mikroprozessor selbst gesetzt oder zurückgesetzt wird, andererseits kann dies auch vom Programm aus mit speziell dafür gedachten Befehlen wie SEC und CLC erfolgen. Dies ist bei den einzelnen Flags unterschiedlich. Manche werden bei bestimmten Operationen automatisch gesetzt oder gelöscht, für andere gibt es spezielle Befehle.

Das N-Flag wird grundsätzlich von fast jedem Assemblerbefehl beeinflußt und nimmt den Zustand des achten Bits an. Bei zum Beispiel LDA ist das eben das achte Bit des Akkumulators, bei LDX das des X-Registers, bei CPY das des Y-Registers usw. Der Zustand des N-Flags bezieht sich also immer auf die laufende Operation.

Im einzelnen wird das N-Flag von folgenden Befehlen gesetzt bezw. rückgesetzt:

ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA, TYA.

Spezielle Befehle zur Beeinflussung des N-Flags gibt es nicht. Somit gibt das N-Flag immer den Status der letzten Datentransferoperation an.

V – Überlauf in das Vorzeichen (overflow)

Das V-Flag ist praktisch nur dann von Bedeutung, wenn bei arithmetischen Operationen wie Addition oder Subtraktion das achte Bit eines Bytes als Vorzeichenbit betrachtet wird. Das V-Flag hat in diesem Fall dieselbe Bedeutung wie üblicherweise das Carry-Flag.
Befehle, die das V-Flag beeinflussen, sind: ADC, BIT, PIP, RTS, SBC und CLV („clear v-flag“, „lösche V-Flag“).

B – Break-Befehl (break command)

Dieses Bit ist wichtig bei der sog. „Interrupt-Steuerung“. Es wird ausschließlich von dem Befehl BRK gesetzt und zeigt an, daß ein „Software-Interrupt“ stattgefunden hat.

D – Dezimalbetrieb (decimal mode)

Der Mikroprozessor kann auch im Dezimalbetrieb arithmetische Operationen ausführen. SED (für „set decimal mode“) schaltet Dezimalbetrieb ein. CLD (für „clear decimal mode“) schaltet wieder aus. Wir haben bisher ausschließlich im Binärbetrieb gearbeitet, was der „natürliche“ Zustand des Prozessors ist.

I – Interrupt sperren (interrupt disable)

„Interrupt“ heißt übersetzt „Unterbrechung“ und bedeutet hier eine Unterbrechung der normalen Tätigkeit des Prozessors, im allgemeinen nur für eine ganz kurze Zeitspanne. Ein typisches Beispiel für Interrupt-Betrieb: Alle 1/60 Sekunden wird der normale Prozessorablauf per Interrupt unterbrochen, um die Tastatur und den Bildschirm zu „bedienen“, d. h. Tastendrücke zu erkennen und ggf. auf dem Bildschirm das entsprechende Zeichen darzustellen.

Man unterscheidet zwischen NMI- und IRQ-Interrupt. Das I-Flag des Statusregisters bezieht sich ausschließlich auf den IRQ-Interrupt.

/ – Null-Ergebnis (zero result)

Das Z-Flag wird – ähnlich wie das N-Flag – bei jeder Datentransfer- oder Berechnungsoperation automatisch beeinflußt. Es wird gesetzt, wenn das Ergebnis der Operation 0 ist. Ist das Ergebnis ungleich 0, so wird das Z-Flag gelöscht.

Das Z-Flag wird von genau den gleichen Assemblerbefehlen gesetzt bzw. rückgesetzt wie das N-Flag. Für beide gibt es keine speziellen Befehle zur Beeinflussung.

C – Carry-Bit (carry)

Das Carry-Flag stellt eine Art neuntes Bit des Akkumulators dar. Wir haben ausführlich darüber gesprochen bzw. werden die noch nicht besprochenen Anwendungen kennenlernen.

Beeinflußt wird das Carry-Bit von: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP. ROL, ROR, RTI, SBC, SEC.

Es ist nicht notwendig, daß Sie sich schon jetzt die Bedeutung aller Bits des Statusregisters merken. Einprägen sollten Sie sich jedoch die Funktionen von Z- und N-Flag. Beide sind für das nun folgende Thema Bedingte Verzweigungen von recht hoher Bedeutung.

Es gibt in Assembler insgesamt acht sog. „Bedingte Sprungbefehle“. Sie heißen wie folgt:

BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

Sie nehmen ausnahmeslos auf einzelne Flags des Statusregisters Bezug. Der Sprung zu einer angegebenen Adresse erfolgt in Abhängigkeit davon, ob bestimmte Statusbits gesetzt sind oder nicht.

Wir wollen uns das Beispielprogramm aus dem letzten Kapitel noch einmal vornehmen und daran die BNE-Anweisung exemplarisch erklären.

```
5000 LDA     #0        ;Akku mit 0 laden
5002 LDX     #0        ;X-Register mit 0 laden
5004 STA  32768,X      ;Akku in 32768+X-Reg. speichern
5007 CLC               ;Carry-Flag löschen
5008 ADC     #1        ;Akkuinhalt um 1 erhöhen
5010 INX               ;Inhalt des X-Register um 1 erhöhen
5011 CPX     #0        ;X-Register mit 0 vergleichen
5013 BNE   5004        ;wenn nicht 0, Sprung zu 5004
5015 RTS               ;andernfalls (gleich 0) Ende.
```

(Bedenken Sie, daß die STA-Anweisung wiederum an den jeweiligen Computertyp angepaßt werden muß; Bildschirmanfang.)

„BNE" steht für „branch on not equal", „verzweige wenn nicht gleich (null)"; oder exakt gesagt, „verzweige wenn das Z-Flag nicht gesetzt ist". BNE fragt also auf eine ganz bestimmte Bedingung des Statusregisters ab – in diesem Fall, ob das Z-Flag gesetzt ist oder nicht. Sehen Sie an dieser Stelle noch einmal nach, wozu genau das Z-Flag dient.

Alle Befehle zu Bedingten Verzweigungen lauten „Branch on...", weswegen sie auch einheitlich „Branch-Befehle" genannt werden. Ist die Bedingung entsprechend dem Statusregister erfüllt, so wird zu der mit dem Branch-Befehl angegebenen Adresse verzweigt. Daß heißt, die Programmausführung wird ab eben dieser Adresse fortgesetzt. Voraussetzung dafür ist natürlich, daß sich ab dieser Adresse gültiger Assemblercode und nicht zum Beispiel irgendwelche Daten befinden. Anderenfalls kann es zu einem völlig unkontrollierten Verhalten des Prozessors kommen. Sie wissen ja, es ist Ihre Aufgabe als der Programmierer, Daten und Programme auseinanderzuhalten, der Prozessor selbst ist nicht dazu in der Lage.

Bei BNE „entscheidet" also das Z-Flag darüber, ob der Sprung erfolgt oder nicht. Wie wir besprochen haben, wird das Z-Flag von Datentransfer- oder Berechnungsoperation beeinflußt; so auch von CPX (Vergleichsanweisungen fallen unter Berechnungsoperationen). In unserem Beispiel beeinflußt folglich die CPX-Anweisung direkt vor dem BNE-Befehl das Z-Flag. Ergibt der Vergleich, daß der Inhalt des X-Registers mit dem angegebenen Wert (im Beispiel 0) übereinstimmt, dann wird das Z-Flag gesetzt, andernfalls zurückgesetzt. Und diese Entscheidung wirkt sich direkt auf den nachfolgenden BNE-Befehl aus.

Andere Branch-Befehle reagieren auf andere Flags des Statusregisters, wir kommen gleich darauf zu sprechen.

Jetzt wollen wir einen kleinen „Trick" anwenden, der Ihnen zeigen soll, wie flexibel die Programmierung in Assembler ist. Wir nehmen nämlich jetzt die CPX-Anweisung komplett aus unserem Programm – und es wird genauso funktionieren, wie vorher.

Unser gekürztes Programm sieht wie folgt aus:

```
5000 LDA      #0          ;Akku mit 0 laden
5002 LDX      #0          ;X-Register mit 0 laden
5004 STA 32768, X         ;Akku in 32768+X-Reg, speichern
5007 CLC                  ;Carry-Flag löschen
5008 ADC      #1          ;Akkuinhalt um 1 erhöhen
5010 INX                  ;Inhalt des X-Registers um 1 erhöhen
5011 BNE  5004            ;wenn nicht 0, Sprung zu 5004
5013 RTS                  ;andernfalls (gleich 0) Ende
```

Warum funktioniert nun das Programm in dieser Form ohne CPX-Anweisung?

Die Erklärung ist eigentlich recht einfach. Zur Erinnerung: Das Z-Flag wird von jeder Datentransfer- oder Berechnungsoperation beeinflußt. Eine Berechnungsoperation ist aber auch INX. Folglich setzt auch INX das Z-Flag, falls das Ergebnis der Operation – der Inhalt des X-Registers – gleich 0 ist. Der Effekt in Bezug auf das Z-Flag ist somit exakt derselbe wie bei CPX wobei ein Befehl eingespart wird.

Der hier aufgezeigte Fall kommt abgewandelt recht häufig vor. Eine Branchenanweisung folgt nicht unbedingt einer Vergleichsanweisung (CMP, CPX oder CPY), sondern direkt auf einen anderen Befehl, der eben das betreffende Flag entsprechend beeinflußt.

Sind Sie soweit klargekommen? Gut, dann wollen wir nun gleich noch ein bißchen weitergehen. Wie Sie sehr wohl wissen, existieren ja außer BNE noch sieben weitere Befehle zur bedingten Verzweigung. Sie alle bewirken Sprünge im Programmablauf in Abhängigkeit davon, ob bestimmte Bits (Flags) des Statusregister gesetzt sind oder nicht. Wir haben hier alle Branch-Befehle mit ihren Bedeutungen vollständig aufgelistet.

BCC - „branch on carry clear“
Sprung bei gelöschtem Carry-Flag

(Nach CMP/CPX/CPY auch als Vergleich auf 'kleiner als'.)

BCS - „branch on carry set“
Sprung bei gesetztem Carry-Flag

(Nach CMP/CPX/CPY auch als Vergleich auf 'größer/gleich'.)

BEQ - „branch on equal (zero)“
Sprung bei gesetztem Z-Flag (Zero-Flag)
(Ergebnis der letzten Operation gleich Null)

BNE - „branch on not equal (zero)“
Sprung bei gelöschtem Z-Flag (Zero-Flag)
(Ergebnis der letzten Operation ungleich Null)

BMI - „branch on minus“
Sprung bei gesetztem N-Flag (Negativ-Flag)

BPL - „branch on plus“
Sprung bei gelöschtem N-Flag (Negativ-Flag)

BVC - „branch overflow clear“
Sprung bei gelöschtem V-Flag

BVS - „branch on overflow set“
Sprung bei gesetztem V-Flag

Schauen Sie die Bedeutungen der einzelnen Flags des Statusregisters nochmals nach, wenn Sie sie nicht mehr alle wissen. Die durchweg am häufigsten benutzten bedingten Sprunganweisungen sind BNE und BEQ. Wir werden im nächsten Kapitel eine ganze Reihe von Beispielprogrammen kennenlernen und besprechen, damit Sie mehr Erfahrung im Umgang mit den neuen Assemblerbefehlen bekommen.

Damit sie das aber verstehen können, müssen wir uns nun wie angekündigt mit der sogenannten „relativen Adressierung“ beschäftigen. Diese Adressierungsart ist ausschließlich bei den Branchenbefehlen möglich, diese wiederum können nur relativ adressieren.

Nehmen wir uns zur Erleichterung ein Beispiel zu Hilfe:

```
Programm:                        Byte-Folge:

 .
5001  INX                        232
5002  BNE  5007                  208    3
5004  STX    22                  134   22
5006  RTS                         96
5007  LDA  #255                  169  255
   . . .
```

Dies soll einen Ausschnitt eines umfangreicheren Programms darstellen. Es ist lediglich zur Veranschaulichung gedacht, nicht zum Eintippen und Ausprobieren.

Zum Ablauf: Das X-Register wird um eins erhöht (INX). Wird es dadurch gleich Null, so wird das Z-Flag gesetzt, andernfalls wird das Z-Flag gelöscht (wir sind immer noch bei INX). Nun kommt die BNE-Anweisung. Bei gelöschten Z-Flag erfolgt ein Sprung zu Adresse 5007 und die Programmausführung wird dort fortgesetzt. Andernfalls werden die auf BNE folgenden Befehle 'STX 22' und RTS ausgeführt. Nebenbei: STX ist „zeropageadressiert“. RTS beendet – wie Sie wissen – das Programm. Soweit also nichts Neues für uns. Das Beachtenswerte daran ist aber, wie die BNE-Anweisung in zwei Bytes abgespeichert wird (Werte 208 und 3). BNE ist ein 2-Byte-Befehl.

Nach Ihrem bisherigen Kenntnisstand könnten Sie erwarten, daß BNE ein 3-Byte-Befehl wäre, ein Byte für das Befehlswort selbst, zwei Bytes für die Adresse als LSB und MSB. BNE ist aber ein 2-Byte-Befehl. Und da das Befehlswort in jedem Fall ein Byte benötigt, wird also die Adresse in nur einem einzigen Byte abgespeichert. Wie kann das vor sich gehen?

Sie wissen, ein Byte kann nur Werte von 0 bis 255 beinhalten, soviel wie eben mit acht Bits darzustellen sind. Wie also kann ein einzelnes Byte die Adresse 5005 speichern?

Die Erklärung: Wenn unser Assemblersystem ...

5002 BNE 5007

...ausgibt, dann deswegen, weil es „mitdenkt". Die Sprungadresse 5007 ist als solche nämlich gar nicht abgespeichert. Vielmehr wird sie von jedem guten Assemblersystem errechnet. Abgespeichert in dem einen zur Verfügung stehenden Byte ist lediglich, um wieviele Bytes gesprungen wird.

Sehen wir uns noch einmal die Byte-Folge an:

Programm:	Byte-Folge:
5002 BNE 5007	208 3

Die 208 ist der Assemblercode für das Befehlswort BNE. Und die 3 bedeutet, daß bei erfüllter Bedingung um drei Bytes nach vorne gesprungen werden soll. Der Prozessor hat zum Zeitpunkt des Sprungs schon die BNE-Anweisung „verlassen" und steht bereits auf dem folgenden Befehl (im Beispiel STX in Speicherstelle 5004). Daher bewirkt die „plus 3" einen Sprung nach vorne an Adresse 5007. Die Rechnung: 5003 (an dieser Adresse steht die 3) plus 1 (da der Prozessor schon am Anfang des nächsten Befehls steht) plus 3 ist gleich 5007.

Jetzt verstehen Sie auch den Begriff „relative Adressierung". Es wird angegeben, um wieviele Bytes relativ zur aktuellen Position verzweigt werden soll. Die eigentliche Adress wird erst errechnet, sie ist nicht direkt im Befehl selbst gespeichert (5003 plus 1 plus 3 ist gleich 5007). Man nennt die gespeicherte „Relativ-Zahl", die zur aktuellen Position plus 1 addiert werden muß (in unserem Beispiel die 3), um die Sprungadresse zu erhalten, auch „Offset".

Jedes gute Assemblersystem akzeptiert bei der Eingabe die Angabe der wirklichen Adresse und errechnet sich daraus den Offset, der in die betroffene Speicherzelle übernommen werden muß. Andererseits wird ein guter Assembler bei der Ausgabe immer auch die wirkliche („absolute") Adresse ausweisen (in unserem Beispiel also die 5007).

Aber auch wenn die Umrechnung automatisch vor sich geht, sollten Sie sie dennoch verstehen. Die folgende Darstellung soll Ihnen noch einmal vor Augen führen, wie ein Offset zu einem Branch-Befehl errechnet wird.

```
1000  BNE
1001  ???   ------      Offset=
1002                              0
1003                              1
1004                              2
1005                              3
1006                              4
1007                              5
1008                              6
 .                                .
 .                                .
1129                            127
```

Sie sehen, ein Offset von 0 bewirkt die Fortsetzung mit dem direkt auf die Branch-Anweisung folgenden Befehl. Andere Offsets verzweigen zu anderen Adressen. Beachten Sie, daß auf diese Weise auch zu Adressen gesprungen werden kann, bei denen gar kein gültiger Assemblerbefehl steht, sondern zum Beispiel ein Argument oder andere Daten. Es ist Ihre Aufgabe als Programmierer, dafür zu sorgen, daß das nicht passiert. Sie wissen ja: Es gibt keinen grundsätzlichen Unterschied zwischen Programm und Daten.

Das Schaubild zeigt auch, daß bei einer Branch-Anweisung um maximal 127 Bytes nach vorne verzweigt werden kann. Diese Einschränkung spielt zwar bei den meisten Anwendungen keine Rolle, aber wenn man sie nicht kennt, kann einen der Versuch, einen weiteren Sprung zu programmieren, zur Verzweiflung bringen. Sie sollten sich daher diesen Punkt merken.

Woher kommt die Einschränkung auf 127 Bytes nach vorne? Nun, in der einen Speicherzelle, die für den Offset zur Verfügung steht, kann eine Zahl von 0 bis 255 stehen. Ein Teil dieses Bereiches wird für Vorwärtssprünge genutzt – nämlich der von 0 bis 127 – der Rest für Rückwärtssprünge. Schließlich muß es ja auch möglich sein, auf vorangehende Adressen zu verzweigen. Erweitern wir unser Schaubild.

```
 872                            128
 .                                .
 .                                .
 998                            252
 999                            253
1000  BNE                       254
1001  ???   ------      Offset= 255
1002                              0
 .                                .
 .                                .
1129                            127
```

Die Graphik zeigt auch: Ein Offset von 254 führt zu einer Endlosschleife, da immer wieder auf den Branch-Befehl selbst gesprungen wird. Ein Offset von 255 bewirkt im allgemeinen den „Absturz“ des Prozessor, da auf die Offsetadresse selbst verzweigt wird, die keinen gültigen Assemblerbefehl enthält.

Alle hier gemachten Erklärungen gelten natürlich nicht nur für BNE, sondern auch für alle anderen Branch-Anweisungen.

Um Ihnen die Möglichkeit zu geben, zu überprüfen, ob Sie die Berechnung eines Offsetwertes auch voll verstanden haben, folgt hier ein kleiner Test.

```
Programm:                  Byte-Folge:

5000 LDA    #0             169   0
5002 LDX    #0             162   0
5004 STA 32768,X           157   0 128
5007 CLC                    24
5008 ADC    #1             105   1
5010 INX                   232
5011 BNE 5004              208 ...
5013 RTS                    96  .
```

Sie kennen dieses Programm bereits aus Kapitel 6. Zum Ablauf muß die Adresse 32768 wieder an den jeweiligen Computertyp angepaßt werden. Uns – genauer gesagt Sie – soll aber jetzt nur interessieren, welcher Offsetwert in Speicherzelle 5012 abgelegt wird bei dem angegebenen BNE-Befehl. Füllen Sie also den durch „... “ freigelassenen Platz aus. Sollte Ihnen das schwerfallen, lesen Sie in Ihrem eigenen Interesse nochmals die letzten Seiten durch.

Wie Sie gelernt haben, ist es mit bedingten Verzweigungsanweisungen nur möglich, um 127 Bytes vorwärts bzw. 128 Bytes rückwärts zu springen. In BASIC wäre das so, als wenn man bei IF-THEN-Anweisungen nur um 127 bzw. 128 Programmzeilen nach vorne bzw. hinten verzweigen darf. Wie zu (fast) jeder Einschränkung existieren natürlich auch Tricks, diese zu umgehen. (Man kann zum Beispiel einen Branchbefehl auf einen weiteren Branchbefehl zeigen lassen, was die Sprungweite verdoppelt). Wir wollen aber hier gar nicht diese Tricks besprechen, sondern den Vorteil, den die relative Adressierung bei genauerem Hinsehen bietet.

In Assembler kommt häufig die Forderung auf, daß ein Programm „relokatibel“ sein soll. Gemeint ist damit, daß das Programm funktionieren soll, unabhängig davon, in welchem Adressbereich es gespeichert ist. Und das ist manchmal gar nicht unproblematisch. Nehmen wir an, Sie schreiben ein Assemblerprogramm, beginnend aber Adresse 10000 und endend mit Adresse 10500. Da Ihr Programm komplexer ist – immerhin hat es eine Länge von 500 Bytes – enthält es auch eine Vielzahl von Sprüngen innerhalb des Programms.

Dieses Programm soll nun ab Adresse 2000 laufen. Die Sprungbefehle müssen dabei natürlich alle umgestellt werden. Sie erinnern sich an den Befehl JMP („jump", Sprung). Aus ...

10230 JMP 10300

..wird also im Zuge der Umstellung...

2230 JMP 2300

Bei dieser Programmierung muß also bei einer Verschiebung in einen anderen Adressbereich jeder einzelne Sprungbefehl umgerechnet werden. Ein gutes Assemblersystem bietet Befehle, die dies vereinfachen, bei ‚T.EX.AS.' sind das im Line-by-Line-Assembler im wesentlichen TRANSFER, UPDATE und CHANGE. Dennoch bleibt der Umstand als solcher bestehen.

Damit sind wir auch bereits beim Vorteil der relativen Adressierung. Bei dieser wird – wie Sie wissen – angegeben, um wieviele Bytes vorwärts oder rückwärts gesprungen werden soll. Das aber ändert sich nicht bei einer Verschiebung des Adressbereiches, der Abstand zwischen den einzelnen Anweisungen bleibt ja immer derselbe. Folglich müssen Branch-Befehle grundsätzlich nicht korrigiert werden, egal, ab welcher Adresse das Programm steht. Das wird oft so weit ausgenutzt, daß erst beim Einladen eines Assemblerprograms bestimmt wird, ab welcher Adresse es geladen wird. Ein Beispiel hierfür sind die EXBASIC LEVEL II SOFTMODULE.

Da nur Branch-Anweisungen relativ adressieren können, werden so manchmal anstelle eines JMP-Befehls genommen, nur um das Programm relokatibel zu machen. So kann zum Beispiel...

10230 JMP 10300

..ersetzt werden durch (Beispiel)...

10230 CLC
10231 BCC 10300

Die Kombination CLC („clear carry", „lösche Carry-Flag") und BCC („branch on carry clear", „verzweige bei gelöschtem Carry-Flag") bewirkt in jedem Fall einen Sprung zu Adresse 10300. Da jedoch die absolute Adresse 10300 nur vom Assemblersystem errechnet wird – Sie erinnern sich! – und in Wirklichkeit nur der Offset (im Beispiel 67) gespeichert wird, funktioniert der Sprungbefehl mit BCC in jedem Adressbereich. Bei JMP hingegen wäre eine Umstellung nötig.
Statt CLC/BCC könnten wir auch zum Beispiel SEC/BCS nehmen („set carry", „setze Carry-Flag" und „branch on carry set", „verzweige bei gesetztem Carry-Flag"). Es muß nur dafür gesorgt sein, daß die Bedingung des Branch-Befehls in jedem Fall erfüllt ist, damit es zum Sprung kommt.

Oftmals ist gar kein spezieller Flag-Befehl nötig, weil eines der Status-Flags aufgrund anderer Anweisungen bereits einen bestimmten Zustand hat.

So, damit wollen wir dieses Kapitel beenden. Schließlich können Sie nicht alles auf einmal lernen. Bevor wir zur Zusammenfassung kommen, hier noch die „Auflösung“ des auf Seite 7–19 abgedruckten Tests. Der Offset lautet 247.

Zusammenfassung

Das, was das Argument einer Assembleranweisung bewirkt, ist durch die Adressierungsart festgelegt. Es existieren (jeweils an einem Beispielbefehl dargestellt):

RIS	implizit
ASI	Akkumulator
LDA #argument	unmittelbar
LDA zeropage-adresse	Zero-Page
LDA zeropage-adresse,X	Zero-Page X-indiziert
LDX zeropage-adresse,Y	Zero-Page Y-indiziert
LDA adresse	absolut
LDA adresse,X	absolut X-indiziert
LDA adresse,Y	absolut Y-indiziert
JMP (adresse)	indirekt
LDA (zeropage-adresse,X)	indiziert-indirekt
LDA (zeropage-adresse),Y	indirekt-indiziert
BEQ offset	relativ

Jedes Befehlswort erlaubt eine unterschiedliche genau festgelegte Anzahl von Adressierungsarten. In dem Kapitel wird die Funktion der Adressierungsarten ausführlich erklärt.

Der Mikroprozessor beinhaltet ein Statusregister („Prozessorstatusregister“) mit acht Bits, wovon bisher sieben genutzt werden. Die Bezeichnung der einzelnen Bits lautet wie folgt:

N: negativ result (negatives Ergebnis)
V: overflow (Überlauf)
B: break command (Break-Befehl)
D: decimal mode (Dezimalbetrieb)
I: interrupt disable (Interrupt sperren)
Z: zero result (Null-Ergebnis)
C: carry (Übertrag)

Die Statusbits werden z.T. von der aktuellen Datentransfer- oder Berechnungsoperation gesetzt oder gelöscht, z.T. existieren spezielle Befehle dafür. Die Bedeutung der Statusbits sowie die Möglichkeit zu deren Beeinflussung sind im Text erklärt.

In Abhängigkeit vom Zustand der einzelnen Bits (Flags) des Statusregisters können bedingte Sprungbefehle (sog. „Branch-Befehle“) programmiert werden.

Es existieren die folgenden acht Branch-Befehle:

BCC: Sprung bei gelöschtem Carry-Flag
BCS: Sprung bei gesetztem Carry-Flag
BNE: Sprung bei gelöschtem Zero-Flag
BEQ: Sprung bei gesetztem Zero-Flag
BPL: Sprung bei gelöschtem Negaiv-Flag
BMI: Sprung bei gesetztem Negativ-Flag
BVC: Sprung bei gelöschtem Overflow-Flag
BVS: Sprung bei gesetztem Overflow-Flag

Alle Branch-Befehle sind Zwei-Byte-Befehle mit relativer Adressierung. D.h., dem Befehlswort folgt nicht die absolute Sprungadresse, sondern ein Offset, der angibt, um wieviele Bytes vor- bzw. rückwärts gesprungen werden soll. Dadurch sind die Sprünge „relokatibel“, d.h. unabhängig vom Speicherbereich. Relative Adressierung ist um maximal 128 Bytes rückwärts und 127 Bytes vorwärts möglich.

Die Berechnung des Offsets wird im Kapitel erklärt, sie wird von jedem guten Assemblersystem selbständig vorgenommen.

Kapitel 8: Das vierte Programm - NOP, TAX, TAY, TXA, TYA, TSX, TXS, JSR, RTS.

Nachdem Sie im letzten Kapitel die Grundlagen für eine ganze Reihe neuer Programmiertechniken kennengelernt haben, wollen wir diese nun in lauffähige Programme umsetzen.

Die Aufgabenstellung für unser erstes Programm dieses Kapitels lautet wie folgt. Es soll geprüft werden, ob die ersten drei Buchstaben des Bildschirms „ABC“ sind und im Falle dies zutrifft, sollen die Buchstaben eine Zeile tiefer kopiert werden.

Da wieder einmal mit dem Bildschirm „gespielt“ wird, müssen Sie die Bildschirmadressen des folgenden Listings einmal mehr an Ihren Computertyp anpassen. Bei Farbcomputern sind die üblichen Umschaltungen der Bildschirmfarbe vor der Programmausführung notwendig. Und so sieht unser Programm aus (Version für CBM 8000):

```
5000  LDA  32768    ; Akku mit 1. Zeichen laden
5003  LDX  32769    ; X-Reg. mit 2. Zeichen laden
5006  LDY  32770    ; Y-Reg. mit 3. Zeichen laden
5009  CMP     # 1   ; Akkuinhalt gleich „A“?
5011  BNE   5030    ; nein, Sprung zu Programmende
5013  CPX     # 2   ; X-Reg. gleich „B“?
5015  BNE   5030    ; nein, Sprung zu Programmende
5017  CPY     # 3   ; Y-Reg. gleich „C“?
5019  BNE   5030    ; nein, Sprung zu Programmende
5021  STA  32848    ; „A“ an 1. Position, 2. Zeile
5024  STX  32849    ; „B“ an 2. Positon, 2. Zeile
5027  STY  32850    ; „C“ an 3. Position, 2. Zeile
5030  RTS           ; Programmende
```

Die „;“-Erklärungen sollten eigentlich ausreichend für Sie sein, um das Programm zu verstehen, wenn Sie die folgenden Punkte beachten:

Die im Programm verwendeten BSC-Codes 1, 2 und 3 stehen für die ersten drei Buchstaben des Alphabets, wie sie ohne Benutzung der SHIFT-Tasten in den Computer eingegeben werden. Die Darstellung erfolgt je nach Darstellungsmodus klein (Textmodus) oder groß (Graphikmodus).

Die Bildschirmadressen 32848, 32849 und 32850 bezeichnen bei dem Computer Commodore CBM 8000 die ersten drei Positionen der zweiten Bildschirmzeile. Bei der Anpassung an andere Computer muß die Anzahl der Zeichen pro Bildschirmzeile berücksichtigt werden. Der CBM 8000 hat 80 Zeichen/Zeile; die Rechnung dazu 32768 plus 80 ist gleich 32848. Bei anderen Computern muß u.U. nur 40 oder gar 22 addiert werden, um den richtigen Wert zu erhalten.

Es stellt sich die Frage, warum nach den Vergleichsbefehlen (CMP, CPX, CPY) ausgerechnet mit BNE („branch on not equal“) verzweigt wird. Schließlich wird ja gar nicht mit Null verglichen. Die Antwort darauf liegt in der Art und Weise, in der die Vergleichsoperationen stattfinden.
Intern wirkt CMP wie folgt: Das zu CMP angegebene Argument wird von dem Inhalt des Akkumulators subtrahiert. Das Ergebnis der Subtraktion wird nirgendwo gespeichert, aber die Flags des Statusregisters werden entsprechend diesem „Scheinergebnis“ gesetzt bzw. gelöscht. Ist das Ergebnis gleich Null, wird also das Z-Flag gesetzt, andernfalls wird es zurückgesetzt (gelöscht). Genau das Z-Flag wird aber von BNE bzw. BEQ berücksichtigt. Ist in unserem Programm das Argument zu CMP nicht gleich dem Akkumulatorinhalt, so ergibt sich ein Wert ungleich 0 und es erfolgt ein Sprung zum Programmende. CPX und CPY verfahren analog mit dem X- und Y-Register.

Jetzt verstehen Sie auch, warum wir bisher die Vergleichsanweisungen dem Bereich Berechnungsoperationen zugeordnet haben. Intern im Prozessor findet eine Subtraktion statt.

Diese Subtraktion ist auch dafür verantwortlich, daß nach CMP/CPX/CPY mit BCS bzw. BCC auf ‚größer/gleich‘ bzw. ‚kleiner als‘ abgefragt werden kann (Seite 7-15). Denken Sie einmal darüber nach.

Gestartet wird das Programm ab Adresse 5000 entweder von BASIC mit „SYS“ bzw. „CALL“ oder von ;T.EX.AS.‘ mit „EX“.

Die Entsprechung unseres Programms in BASIC sieht übrigens wie folgt aus:

```
5000  A= PEEK (32768)
5003  X= PEEK (32769)
5006  Y= PEEK (32770)
5009  IF A()1 THEN 5030
5013  IF X()2 THEN 5030
5017  IF Y()3 THEN 5030
5021  POKE 32848,A
5024  POKE 32849,X
5027  POKE 32850 Y
5030  END
```

Sowohl in BASIC als auch in Assembler ließe sich das Programm auch mit nur einer Variablen bzw. nur dem Akkumulator realisieren. Während dies in BASIC auch sinnvoller wäre, bietet es sich in Assembler geradezu an, für genau drei Buchstaben die drei internen Prozessorregister zu verwenden. Das zeigt einmal mehr, daß die Programmierung in Assembler eine eigene Denkweise gegenüber BASIC erfordert.
Aber auch die Vorgehensweise der Programmierung selbst ist in Assembler meist anders als in BASIC. Da Sie bisher fast ausschließlich fertige Assemblerprogramme eingetippt haben, wird es Ihnen noch nicht so stark aufgefallen sein. Wir wollen jetzt ein Programm mit fortgeschrittenen Möglichkeiten der Programmentwicklung schreiben.

Unser neues Programm soll dazu dienen, einen Speicherbereich darauf zu überprüfen, ob er in Ordnung ist. Dazu wird in alle Speicherzellen des zu testenden Bereichs der Wert 255 geschrieben und anschließend geprüft, ob dieser Wert in allen Zellen noch vorhanden ist. Bei 255 werden alle acht Bits jeder Speicherzelle auf 1 gesetzt, fällt eines aus – gleich aus welchem Grund – so verändert sich der Wert. Dieses Verfahren ist natürlich nur für RAM-Speicher geeignet, nicht für ROM. (Sie erinnern sich an den Unterschied – wir haben in Kapitel 5 darüber gesprochen.)

Um das Programm noch übersichtlich zu halten, wollen wir uns darauf beschränken, lediglich einen Bereich von 256 Bytes zu prüfen. Sind alle Speicherzellen in Ordnung, so soll auf dem Bildschirm in der linken oberen Position ein „J“ ausgegeben werden, andernfalls ein „N“.

Die zwei dafür wichtigen Werte – der Anfang des zu testenden Bereichs und Bildschirmanfang – wollen wir bereits festlegen, bevor wir mit der eigentlichen Programmierung beginnen. Geben Sie dazu in ‚T.EX.AS.‘ folgendes ein:

```
BEREICH 6000
```

Der zu prüfende RAM-Bereich soll ab Adresse 6000 beginnen. Es spielt bei der Eingabe keine Rolle, ob vor dem Wort „BEREICH“ eine Adresse steht oder nicht, diese wird sowieso ignoriert. Beachten Sie dabei, daß das Wort „BEREICH“völlig willkürlich gewählt ist, sie können auch ruhig ein anderes nehmen. Wichtig ist allerdings, daß Sie das Wort OHNE Benutzung der SHIFT-Tasten eingeben, egal, ob es dann in Klein- oder Großbuchstaben auf dem Bildschirm erscheint.

Die obige Anweisung legt fest, daß zukünftig statt der Zahl 6000 das Wort „BEREICH“ als Argument zu einer Assembleranweisung eingegeben werden darf. Man sagt, „BEREICH“ ist ein „Label“. Manchmal wird „Label“ mit dem Ausdruck „Marke“ oder „Programmmarke“ übersetzt, wir werden aber in diesem Buch „Label“ sagen, da dies am gebräuchlichsten ist. Allgemein ist ein Label ein Wort, das als Ersatz für eine Zahl steht. Der Grund zur Verwendung von Labels: Es ist viel einfacher, sich ein Wort zu merken, das ja sinnfällig gewählt werden kann, als eine Zahl.

Meist unterscheidet man zwischen Label-Assemblersystemen und Direktassemblern. Beide haben ihre Vor- und Nachteile. Wenn Sie mit ‚T.EX.AS.‘arbeiten, entfällt diese Unterscheidung praktisch, da ‚T.EX.AS.‘ sowohl im Direktassemblerteil – mit dem wir bisher ausschließlich gearbeitet haben – Labels zuläßt, als auch einen eigenen Label-Assembler (auch „Editor-Assembler“) beinhaltet.

Legen wir nun also den zweiten Label unseres Speichertestprogramms fest, die Anfangsposition des Bildschirms. Das geschieht natürlich wieder je nach Computertyp verschieden.

Wir wollen das Label „Bild" nennen (Eingabe wiederum ohne SHIFT-Tasten!). Für die Commodore CBM-Computer (2000 bis 8000) sieht das wie folgt aus:

Bild 32768

Wählen Sie die für Ihren Computertyp richtige Adresse.

Hier ist nun das Programm abgedruckt, so wie Sie es eingeben. Die Labels werden sofort nach Drücken der RETURN-Taste durch die entsprechenden Zahlen ersetzt. Die Leerzeilen, Trennlinien („---") und „;"-Kommentare werden natürlich nicht mit eingegeben, sie dienen lediglich zur Erleichterung der Erklärung des Programms.

```
; Speicherbereich mit 255 vollschreiben

5000 LDX    #0           ; Zähler (X-Reg.) mit 0 laden
5002 LDA   #255          ; Akku mit 255 als Testwert laden
5004 STA  Bereich,X      ; 255 abspeichern
5007 INX                 ; Sprung zu 5004, solange X von 0
5008 BNE   5004          ; bis 255 zählt (INX/BNE)
------------------------------------------------------------

; Speicherbereich mit 255 vergleichen

                         ; noch aus vorherigem Programmteil:
                         ; Akku ist 255, X-Reg. ist 0
5012 CMP  Bereich,X      ; Akku mit Speicher vergleichen
5015 BNE   5026          ; nicht korrekt, Sprung zu 5026
5017 INX                 ; Sprung zu 5012, solange X von 0
5018 BNE   5012          ; bis 255 zählt (INX/BNE)
                         ; Test ist korrekt verlaufen
------------------------------------------------------------

; „J" auf den Bildschirm schreiben

5020 LDA   #10           ; Akku mit „J" laden (BSC-Code 10)
5022 STA  Bild           ; und auf den Bildschirm bringen
5025 RTS                 ; Programmende
------------------------------------------------------------

; „N" auf den Bildschirm schreiben

5026 LDA   #14           ; Akku mit „N" laden (BSC-Code 14)
5028 BNE  5022           ; Sprung zu 5022 (immer!)
```

Schon der Programmumfang zeigt, daß wir es diesmal mit einem komplexeren Problem zu tun haben. Aber schließlich wollen Sie ja auch Fortschritte machen.

Das Programm wird bei Adresse 5000 gestartet. Damit Sie den Buchstaben „J“ oder „N“ erkennen können, muß bei Farbcomputern u.U. wiederum die Bildschirmfarbe zuvor verändert werden. Probieren Sie das Programm aus. (Und erhalten dabei hoffentlich ein „J“).

Wie Sie dem Listing entnehmen können, gliedert sich unser Speichertestprogramm in vier Funktionseinheiten. Beginnen wir unsere Erklärung mit dem ersten.

Die ersten 256 Bytes des gewählten Speicherbereiches werden mit dem Wert 255 gefüllt – bei unserer Bereichswahl von 6000 bis 6255. Der Akkumulator enthält den zu speichernden Wert 255, das X-Register dient als Zähler von 0 bis 255 in einer Schleife. Obgleich das X-Register zu Anfang mit 0 geladen wird, erhöht INX noch während des ersten Schleifendurchlaufs auf 1. Damit ist die Sprungbedingung BNE (Verzweige, wenn das Ergebnis ungleich 0 ist, Z-Flag gelöscht) erfüllt, und zwar bis zum letzten Mal beim Wert 255. Wird INX das nächste Mal abgearbeitet, so wird das X-Register wieder 0 (Sie wissen, auf 255 folgt 0, 1, 2 usw.). Damit aber ist die BNE-Bedingung nicht erfüllt, denn das Z-Flag wurde gesetzt. Der Speicherbereich ist mit 255 gefüllt und die Programmabarbeitung setzt bei Adresse 5012 fort.

Zu diesem Zeitpunkt beinhalten sowohl Akkumulator als auch X-Register genau bestimmte Werte noch aus der vorherigen Abarbeitung, nämlich der Akkumulator nach wie vor 255 – daran ist ja nichts geändert worden – und das X-Register 0 – denn das war ja genau die Bedingung, um aus der Schleife herauszukommen (BNE): Daher brauchen im zweiten Programmteil Akkumulator und X-Register nicht neu geladen zu werden.

Die Anweisungen von Adresse 5012 bis 5018 bilden wiederum eine Schleife, bei der das X-Register ebenso von 0 bis 255 zählt. Nur diesmal wird der Akkumulatorinhalt (255) nicht mit STA abgespeichert, sondern vielmehr mit CMP verglichen. Wie Sie dem letzten Kapitel unter der Adressierungsart X-indiziert entnehmen können, ist diese Möglichkeit sowohl bei STA als auch bei CMP gegeben. Dadurch, daß das X-Register nacheinander die Werte von 0 bis 255 durchzählt, sprechen STA bzw. CMP auch nacheinander die Speicherstellen BEREICH+0, BEREICH+1, BEREICH+2 usw. bis BEREICH+255 an.

Der Vergleichsanweisung CMP folgt direkt ein BNE-Befehl. Sind die Inhalte von Akkumulator und zu prüfender Speicherzelle nicht gleich – nämlich gleich 255 – so ist die BNE-Bedingung „ungleich 0“ erfüllt und es wird zu Adresse 5026 („N“ ausgeben) verzweigt. Sie erinnern sich: CMP führt intern eine Subtraktion durch und setzt die Flags des Statusregisters entsprechend dem Ergebnis.

Ist X wiederum bei 255 angelangt und zählt noch eins weiter (INX), so wird das X-Register 0, die ‚BNE 5012‘-Bedingung ist nicht mehr erfüllt und die Programmausführung geht bei Adresse 5020 weiter („J“ ausgeben).

Die Ausgabe des Buchstabens „J“ auf dem Bildschirm ist einfach programmiert, Sie sollten das problemlos verstehen. Die „N“-Ausgabe sieht hingegen auf den ersten Blick etwas eigenartig aus. Sie wird von Adresse 5015 (BNE 5026) aus angesprungen, falls eine Speicherstelle nicht mehr die eingeschriebene 255 enthält.

Nun, wir hätten die „N“-Ausgabe auch als ...

```
5026 LDA  #14
5028 STA Bild
5031 RTS
```

.. programieren können. Kürzer aber ist es, wenn wir STA und RTS aus der „J“-Ausgabe mitbenutzen. Dazu ist ein Sprung zu Adresse 5022 nötig. Das wiederum ließe sich wie folgt programmieren:

```
5026 LDA  #14
5028 JMP 5022
```

Die Anweisung „BNE 5022“ erfüllt an dieser Stelle aber genau dieselbe Funktion wie JMP. Da der Akkumulator mit dem Wert 14 geladen wird, ist das Z-Flag in jedem Fall gelöscht (14 ist schließlich ungleich 0), so daß die BNE-Bedingung immer erfüllt ist. Und BNE anstelle von JMP zu verwenden spart nicht nur ein Byte – JMP ist ein 3-Byte-Befehl, BNE ein 2-Byte-Befehl – sondern macht das Programm auch relokatibel. Unser Speichertestprogramm kann an jeder beliebigen Stelle im Speicher stehen, es funktioniert immer – solange Sie nicht versuchen, genau den Speicherbereich zu prüfen, in dem gerade das Prüfprogramm steht.

Ja, damit ist auch die Funktion des zweiten Programms dieses Kapitels erklärt. Wenn Sie Lust haben, können Sie ja einmal probieren, es so auszubauen, daß es möglich wird, auch mehr als 256 Bytes auf einmal zu überprüfen. Das ist deswegen nicht ganz einfach, da das X-Register nur von 0 bis 255 zählen kann, und ebenso auch der Akkumulator und das Y-Register. Aber Sie können ja versuchen, zwei Schleifen, sagen wir mit X- und Y-Register und INX bzw. INY realisiert, zu schachteln. Die Y-Schleife zählt zum Beispiel von 0 bis 3 und innerhalb jedes Schleifendurchlaufs zählt eine X-Schleife von 0 bis 255. Damit lassen sich immerhin 4 mal 256 gleich 1024 Bytes (1KByte) ansprechen.

Es ist auch möglich, den Zähler nicht in einem der Prozessorregister zu realisieren, sondern in einer beliebigen Speicherzelle. Sie haben ja gelernt, 2-Byte-Zahlen zu addieren. Über eine indirekte Adressierungsart läßt sich das ausnutzen.

Nun haben Sie zwar schon gelernt, was ein Label ist und wie man sich damit das Programmieren vereinfachen kann, aber zu einer effektiven Programentwicklung reicht das bei weitem nicht aus. Wir haben bisher ausschließlich mit dem Direktassembler – manchmal auch „Line-by-Line-Assembler“ genannt – gearbeitet. Dabei wird ein eingegebener Befehl direkt in die entsprechenden Assemblercodes umgewandelt, und das Zeile für Zeile („line-by-line“).

Um abschätzen zu können, wann der Einsatz eines Direktassemblers sinnvoll ist und wann nicht, muß man über Vor- und Nachteile Bescheid wissen.

Die Vorteile: Das Programm steht jederzeit zur Verfügung – soweit Sie es eingetippt haben.Sie können das Programm sofort starten, um es auszuprobieren.

Aber die Nachteile sind recht gravierend: Es ist nur schwer möglich, neue Befehle in das Programm einzufügen oder welche zu entfernen. Labels werden – sofern sie überhaupt zugelassen sind – sofort wieder in Zahlen umgewandelt; wenn man sich die Bedeutung aller verwendeten Datenspeicher nicht merken kann, insbesondere bei langen Programmen, muß man sich eine Liste der benutzten Speicherzellen anfertigen. Kommentare zu dem Programm können nicht in das Programm selbst geschrieben werden – denken Sie an die BASIC-Anweisung REM – sondern müssen extern auf Papier festgehalten werden.

Es gibt Programmierer, die auch die Entwicklung hochkomplexer Programme mit einem Direktassembler vornehmen. Das erfordert viel Selbstdisziplin (Dokumentation mitschreiben!) und reichlich Erfahrung sowie ein sehr gutes Assemblersystem. Mit ‚T.EX.AS.‘ ist so etwas möglich. Die meisten Programmierer ziehen aber einen anderen Weg vor.

Wir haben bereits darüber gesprochen, daß es neben Direktassemblern auch Editor-Assembler gibt. Unter einem Editor-Assembler ist folgendes zu verstehen. Alle Assembleranweisungen, welche Sie eintippen, werden nicht gleich in Assemblercode umgewandelt, sondern bleiben vielmehr als Text bestehen. Das bedeutet, auch Labels werden nicht sofort in Zahlen umgesetzt, sondern bleiben ausgeschrieben stehen. Die einzelnen Assembleranweisungen werden mit Zeilennummern versehen, die unabhängig von Speicherzellen sind. Sie können also zum Beispiel Zeilennummern von 100 bis 500 in Zehnerschritten wählen (100, 110, 120 ...). Die dazwischenliegenden Zeilennummern können benutzt werden, wenn Assemblerbefehle eingefügt werden sollen. Das Löschen von Befehlen geschieht durch Eingabe einer „leeren“ Zeilennummer.

Das entspricht im Grunde fast vollkommen der Programmierung in BASIC oder zum Beispiel auch PASCAL, nur die eigentliche Programmiersprache ist natürlich Assembler. Auch das Listen, Ändern etc. wird ähnlich oder sogar genauso wie in BASIC vorgenommen. Kommentare können direkt in den Programmtext geschrieben werden, ähnlich einer REM-Anweisung in BASIC.

Den Teil, der dafür sorgt, daß Sie Eingaben tätigen können und diese auf dem Bildschirm entsprechend dargestellt, geändert, gelöscht etc. werden, nennen wir „Editor". Mit dem Editor wird der Assembler-TEXT erstellt, aber noch kein Assembler-CODE. Man spricht auch von „Source Text" (Quelltext) und „Object Code" (Objekt Code). Die Umwandlung von Quelltext – der bequem zu editieren ist – in den eigentlichen Objekt-Code – nur der wird vom Mikorprozessor verstanden – geschieht in der sog. „Assemblierung". Diese Umwandlung, die bei dem Direktassembler sofort bei jeder Eingabe geschieht, wird also beim Editor-Assembler erst nach Fertigstellung des gesamten Programms vorgenommen. Meist existieren zusätzliche Befehle zur Steuerung der Assemblierung, sog. „Pseudokommandos",. „Pseudo" deshalb, weil sie ausschließlich zur Steuerung des Ablaufs der Assemblierung dienen und nicht vom Prozessor selbst verarbeitet werden können. Es ist üblich, die Pseudokommandos mit einem Punkt am Anfang zu kennzeichnen (zum Beispiel „.BA=5000" und „.END").

Unser letztes Programm als Editortext geschrieben, könnte wie folgt aussehen.

```
100    ;Programm zum Überprüfen eines Speicherbereichs
110    .BA=5000
120    BEREICH=6000
130    BILD=32768
200    ;RAM-Bereich mit 255 füllen
210              LDX #0
220              LDA #255
230    LABEL 1   STA BEREICH,X
240              INX
250              BNE LOOP 1
300    ;RAM-Bereich auf 255 testen
310    LABEL 2   CMP BEREICH,X
320              BNE NEIN
330              INX
340              BNE LABEL 2
400    ;„J" auf dem Bildschirm ausgeben
410              LDA ,J'
420    AUSGABE STA BILD
430              RTS
500    ;„N" auf dem Bildschirm ausgeben
510    NEIN      LDA ,N'
520              BNE AUSGABE
530    .END
```

Sie sehen, Labels bleiben als Wort erhalten, sie werden auch als Zielangaben für Sprungbefehle genutzt („BNE AUSGABE"). Kommentare sind fester Bestandteil des Quelltextes, sie werden mit „;" von gültigem Assemblertext getrennt. Um das Programm ablauffähig zu machen, muß es assembliert werden.

Bedenken Sie dabei: Die Pseudo-Kommandos sind Bestandteil des jeweiligen Assembler-Systems und sie müssen ggf. an Ihren Assembler angepaßt werden.

Jeder Editor-Assembler ist etwas anders zu bedienen. Es kann nicht Aufgabe eines Assembler-Buches sein, darauf im Detail einzugehen. Wenn Sie es bisher noch nicht getan haben, so ist es sicher jetzt an der Zeit, daß Sie das Handbuch zu Ihrem Assemblersystem ausführlich studieren. Ihre Assemblerkenntnisse sind mittlerweile so weit fortgeschritten, daß Sie dabei keine nennenswerten Probleme mehr haben sollten. Das gilt zumindest für das ‚T.EX.AS.'-Handbuch.

Wir wollen unser nächstes Programm zur Übung einmal mit einem Editor-Assembler schreiben. Falls Ihnen ein solcher nicht zur Verfügung steht, können Sie aber auch wie bisher mit einem Direktassembler arbeiten. Die hier benutzten Pseudokommandos beziehen sich auf ‚T.EX.AS.', ändern Sie sie ggf. für Ihr System.

Nachdem wir uns bereits beim letzten Programm Gedanken darum gemacht haben, wie ein Speicherbereich von mehr als 256 Bytes angesprochen werden kann, wollen wir nun dies Problem lösen. Die Aufgabe: Der gesamte Bildschirm soll mit einem bestimmten Zeichen gefüllt werden. Sehen Sie sich das Programmlisting in Ruhe an und überlegen Sie, wie es funktionieren könnte. In der vorliegenden Form ist das Programm für den Commodore CBM 8000 Computer ausgelegt, die notwendigen Änderungen für gängige andere Computertypen sind angefügt.

```
100 .BA=5000
110 ZEICHEN=1
200            LDY  #0            ;--speichert Bildschirmanfang
210            STY  80            ; als LSB und MSB in Adressen
220            LDA  #128          ; 80/81 ab (0+128*256 = 32768)
230            STA  81            ;-----
240            LDA  #ZEICHEN      ;Bildschirmfüllzeichen
300LABEL       STA (80),Y         ;-----speichert Füllzeichen 256
310            INY                ;       mal auf dem Bildschirm,
320            BNE  LABEL         ;--     dann muß MSB erhöht werden
330            INC  81            ;erhöht MSB für Bildschirmanfang
340            LDX  81            ;von 128 bis 136 (X-Reg., damit
350            CPX  #136          ;der Akku „unbeschädigt" bleibt)
360            BNE LABEL          ;-----
400            RTS                ;Programmende
500.END                           ;Ende der Assemblierung
```

Die Adressen 80 und 81 sind willkürlich gewählt und müssen u.U. für andere Computer geändert werden. Wichtig ist, daß diese Adressen vom Computerbetriebssystem selbst nicht laufend benutzt werden. Die Werte 0 und 128 (Zeilen 200 und 220) müssen der Adresse des Bildschirmanfangs angepaßt werden (LSB und MSB). Die Festlegung ZEICHEN in Zeile 110 bestimmt das Zeichen, mit dem der Bildschirm gefüllt wird (von 0 bis 255). „.BA=5000" in Zeile 100 legt fest, in welchen Speicherbereich assembliert wird. Oftmals wird dafür in anderen Assembler-Systemen als ‚T.EX.AS.' auch ‚*=5000' geschrieben. Konsultieren Sie ggf. das Handbuch zu Ihrem Assembler-System.

Hier die notwendigen Änderungen für einige weit verbreitete Computer:

CBM 3000/4000	VC-20 (bis 8K)	VC-20 (über 8K)
	200 LDY # 0	200 LDY # 0
	220 LDA # 30	220 LDA # 16
350 CMP # 132	350 CMP # 32	350 CMP # 18

Commodore 64	Apple II/IIe
200 LDY # 0	200 LDY # 0
220 LDA # 4	220 LDA # 4
350 CMP # 8	350 CMP # 8

Geändert werden sowohl die Adresse des Bildschirmanfangs als auch die Anzahl der auf dem Bildschirm dargestellten Zeichen.

Bei Farbcomputern muß u.U. wiederum die Bildschirmfarbe vor Ausführung des Programms geändert werden.

Kommen wir nun zu der Erklärung des Programms. Es besteht im wesentlichen aus zwei Teilen, dem ersten bis Zeile 240, der die Labels und Speicherzellen mit Werten vorbelegt und dem zweiten ab Zeile 300, der die Schleife bildet, die den Bildschirm füllt.

Im ersten Teil werden die Speicherzellen 80 und 81 mit den Werten gefüllt, die als LSB und MSB interpretiert auf die erste Position des Bildschirms zeigen.

Nun wird in einer Programmschleife von Zeile 300 bis 320 das Y-Register von 0 bis 255 gezählt. Die Speicherstellen 80 und 81 werden indirekt-adressiert angesprochen. Der Akkumulatorinhalt wird also nicht etwa in Adresse 80+Y abgespeichert, sondern in der Adresse, auf die eben 80/81 (LSB/MSB) plus Y zeigen. Zählt das Y-Register über 255 hinaus, so beginnt es wieder bei 0 und die Bedingung BNE ist nicht mehr erfüllt, die Programmausführung wird bei Zeile 330 fortgesetzt.

In Zeile 330 wird das höherwertige Byte (MSB) der in 80/81 abgelegten Adresse um eins erhöht. Da aber das MSB immer mit 256 multipliziert wird, bedeutet eine Erhöhung des MSB um eins eine Erhöhung um 256. Auf diese Weise geht das MSB praktisch in 256er-Schritten über den gesamten Bildschirm. Wieviele Schritte notwendig sind, hängt von der Bildschirmgröße ab und errechnet sich aus der Differenz, mit dem Adresse 81 in den Zeilen 220/230 geladen wird zu dem CMP-Vergleich in Zeile 350.

Der Rücksprung von Zeile 350 zu Zeile 300 (LABEL) erfolgt so lange, bis der Wert in Speicherzelle 81 mit dem durch CMP spezifizierten übereinstimmt. In diesem Fall nämlich ist der gesamte Bildschirm gefüllt und das Programm wird beendet.

Wenn Sie das Programm – wie vorgeschlagen – mit einem Editor-Assembler geschrieben haben, so muß es nun erst assembliert werden, bevor Sie es starten können. In ‚T.EX.AS.' geschieht dies mit dem Makro-Assembler-Teil (ein Makro-Assembler ist ein hochentwickelter Editor-Assembler). Gestartet wird das Programm nach der Assemblierung ab Adresse 5000.

Wir haben bereits darauf aufmerksam gemacht, daß bei manchen Farbcomputern, speziell Commodore 64 und VC-20, aber auch anderen, vor Ausführung eines Programms, das Zeichen auf den Bildschirm bringt, die Bildschirmfarbe selbst geändert werden muß. Das liegt daran, daß diese Computer neben dem Bildschirmadressbereich noch einen Farbadressbereich besitzen. Zu jeder Bildschirmposition existiert eine Adresse, in der der Farbwert der betreffenden Stelle abgelegt ist. Die Speicherzellen dieses Farbbereichs aber sind normalerweise mit dem Standardwert belegt, den auch der übliche Bildschirmhintergrund hat.

Die Folge kennen Sie: Die Farbe eines einzelnen Zeichens ist identisch mit der des Hintergrundes, und das Zeichen somit nicht erkennbar. Wir haben bisher immer die Bildschirmfarbe insgesamt geändert, um dem abzuhelfen. Probieren Sie doch einmal, unser zuletzt entwickeltes Programm so zu erweitern, daß es zu jedem Zeichen, das auf den Bildschirm gebracht wird, den Farbcode an die korrespondierende Stelle des Farbspeicher schreibt.
Das Einfügen von Zeilen ist ja mit dem Editor-Assembler kein Problem mehr. Natürlich muß nach jeder Änderung des Quellprogramms eine erneute Assemblierung stattfinden.

Bei Apple zum Beispiel existiert für den Textmodus kein Farbbereich, weswegen dort die Farbanpassung entfallen kann.

Nachdem wir nun an drei Programmen die im letzten Kapitel besprochenen neuen Adressierungsarten sowie bedingten Verzweigungen beispielhaft in der Praxis kennengelernt haben, wollen wir jetzt noch einige ganz neue Assemblerbefehle vorstellen. Beginnen wir mit einem Befehl, der eigentlich gar keiner ist.

Das Befehlswort „NOP" steht für „no operation", übersetzt „keine Operation". NOP ist ein 1-Byte-Befehl. Der Befehl NOP wird vom Mikroprozessor praktisch ignoriert, es wird keinerlei Operation unternommen, sondern zum nächstfolgenden Befehl übergegangen. NOP erscheint zwar auf den ersten Blick völlig sinnlos, ist aber in Wirklichkeit äußerst hilfreich. Dank der NOP-Anweisung ist es nämlich möglich, auch mit einem Direktassembler Befehle innerhalb eines Assemblerprogramms zu „löschen", ohne das gesamte Programm neu schreiben zu müssen. Die zu „löschenden" Befehle werden einfach durch NOP-Anweisungen ersetzt.

Ein Beispiel zeigt Ihnen die Verwendung von NOP. Nehmen wir folgenden Ausschnitt aus einem umfangreicheren Programm:

```
5320 LDA 3255
5323 BNE 5400
5325 STA 3256
```

Sie stellen nun im Laufe der Programmentwicklung fest, daß die BNE-Anweisung an dieser Stelle falsch ist. Nun können wir ja den BNE-Befehl nicht einfach „herausnehmen", sondern irgendetwas muß ja in den beiden Speicherzellen 5323 und 5324 stehen. Es bestünde natürlich die Möglichkeit, die STA-Anweisung und alle nachfolgenden Anweisungen um zwei Bytes nach vorne zu verschieben. Das jedesmal zu tun, wäre aber sehr umständlich, außerdem hat gar nicht jedes Assemblersystem die dafür nötigen Befehle (schließlich müssen alle nicht-relativen Sprung adressen bei einer Verschiebung nachgestellt werden!).

In so einem Fall bietet es sich an, die beiden Adressen 5323 und 5324 mit NOP zu belegen, wodurch der Programmteil dann so aussieht:

```
5320 LDA   3255
5323 NOP
5324 NOP
5325 STA   3256
```

Beim Arbeiten mit einem Editor-Assembler bzw. Makro-Assembler hingegen ist NOP praktisch überflüssig.

Außer NOP wollen wir in diesem Kapitel als neue Assemblerbefehle noch sämtliche „Transfer-Anweisungen" kennenlernen. Es gibt insgesamt sechs davon, alle beginnen mit dem Buchstaben „T" für „Transfer", gefolgt von den Angaben der Register, deren Inhalte von dem Transfer betroffen sind. Alle Transfer-Befehle sind 1-Byte-Befehle mit der Adressierungsart „implizit".

TAX	„transferiere Akku nach X-Register"
TAY	„transferiere Akku nach Y-Register"
TXA	„transferiere X-Register nach Akku"
TYA	„transferiere Y-Register nach Akku"
TSX	„transferiere Stackpointer nach X-Register"
TXS	„transferiere X-Register nach Stackpointer"

TSX und TXS wollen wir im Moment noch unberücksichtigt lassen, da wir den Begriff „Stackpointer" noch nicht besprochen haben. Wir werden uns im nächsten Kapitel ausführlich damit beschäftigen.

Die Bedeutung der Transfer-Befehle, bis auf TSX und TXS, ist wohl klar. Beachten Sie, daß nur eine Kopie des Inhaltes des jeweiligen Registers in das jeweils andere Register transferiert wird. Im Ursprungsregister bleibt der Wert nach wie vor erhalten.

Ein Beispiel: Im Rahmen eines größeren Programmes sollen Akkumulator sowie X- und Y-Register mit dem Wert der Speicherzelle 826 geladen werden. Mit den bisherigen Mitteln hätten Sie das vermutlich wie folgt programmiert:

```
5320 LDA  826
5323 LDX  826
5326 LDY  826
```

Jetzt würden Sie das – hoffentlich! – einfacher codieren:

```
5320 LDA  826
5323 TAX
5324 TAY
```

Wie Sie wissen, werden auf diese Weise vier Bytes eingespart.

Der Vollständigkeit zuliebe wollen wir auch den Tranfer-Befehlen deren Äquivalente in BASIC gegenüberstellen.

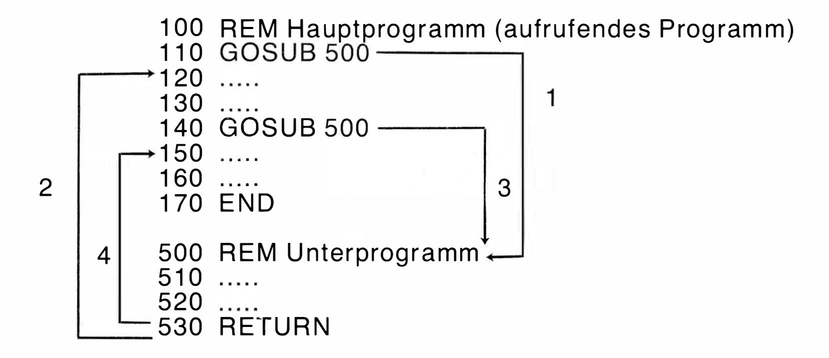
TAX	..entspricht etwa...	X=A
TAY	..entspricht etwa...	Y=A
TXA	..entspricht etwa...	A=X
TYA	..entspricht etwa...	A=Y

Für TSX und TXS existieren in BASIC keine Entsprechungen, wir werden noch darauf zurückkommen. Auf keinen Fall dürfen wir „S=X" oder „X=S" schreiben. Das „S" bei TSX bzw. TXS hat nämlich keineswegs die Bedeutung eines Variablenersatzes, wie das zum Beispiel beim X-Register der Fall ist.

So, die Transfer-Befehle sind so einfach, daß Sie sicherlich keine Probleme haben werden, sie anzuwenden. Wir wollen deswegen auch gar nicht mehr dazu sagen, sondern uns einem anderen – wichtigen – Thema zuwenden.
Es geht um Unterprogrammtechniken. Sollte das Wort „Unterprogramm" für Sie völlig neu sein,ziehen Sie bitte ein geeignetes BASIC-Lehrbuch o.ä. zu Rate. Unterprogrammtechnik ist ein so grundlegendes Gebiet der Programmierung allgemein, daß Sie das am besten in einer Ihnen bereis bekannten Programmiersprache lernen. Wir können und wollen in diesem Buch nicht Programiertechniken als solche lehren, es sei denn, diese sind spezifisch für die Programmierung in Assembler. Und von letzteren haben wir schon genug gelernt und noch zu lernen!

Immerhin, soviel sei zum Thema „Unterprogrammtechnik allgemein" gesagt: Ein Unterprogramm ist ein Teil des Gesamtprogrammes, der von verschiedenen Stellen innerhalb des Programms „aufgerufen" wird. „Aufruf" meint hier, das Unterprogramm wird angesprungen und nach Beendigung des Unterprogramms wird im Gesamtprogramm an der Stelle weitergearbeitet, von der aus das Unterprogramm angesprungen wurde. In BASIC lauten die Befehle dazu GOSUB und RETURN.

Ein Beispiel für eine Unterprogrammstruktur in BASIC:



Der Ablauf ist anhand der Sprungpfeile gut ersichtlich. Das Unterprogramm wird an zwei verschiedenen Stellen des Hauptprogramms (110 und 140) aufgerufen (angesprungen). Dabei „merkt" sich der Computer, von wo aus der Aufruf erfolgt und setzt die Programmausführung nach Beendigung des Unterprogramms (RETURN) mit der nächstfolgenden Anweisung fort, einmal mit Zeile 120, ein andermal mit Zeile 150. Dieses „Merken" wird uns in Assembler noch beschäftigen.

Zunächst wollen wir jedoch die den BASIC-Anweisungen GOSUB und RETURN entsprechenden Assembleranweisungen kennenlernen, die Befehlsworte sind „JSR" und „RTS". „JSR" steht für „jump to subroutine", zu deutsch „Sprung zu Unterprogramm". „RTS" bedeutet „return from subroutine", „Rückkehr aus Unterprogramm".

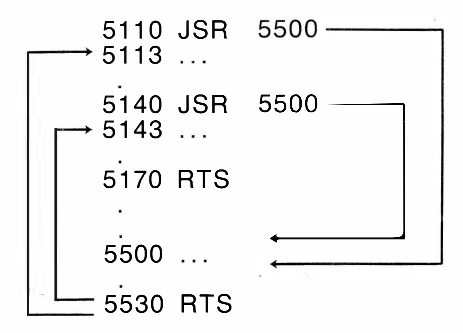
JSR	..entspricht etwa...	GOSUB
RTS	..entspricht etwa...	RETURN

Die Entsprechungen zu BASIC sind recht genau, d.h., JSR kann tatsächlich vollkommen analog zu GOSUB benutzt werden, nur natürlich in Assembler statt in BASIC. Gleiches gilt für RTS. Wir haben in Assembler wie in BASIC die Möglichkeit, Unterprogramme zu schachteln, ein Unterprogramm kann also ein anderes Unterprogramm aufrufen. Die Schachtelungstiefe – sie liegt je nach Anwendung bei ca. 100 bis 120 Unterprogrammebenen – darf auf keinen Fall überschritten werden.

Da JSR ein Sprungbefehl ist, muß als Argument die Adresse angegeben werden, zu der gesprungen werden soll. Das ist die Adresse, ab der das aufzurufende Unterprogramm beginnt. Die Adressierungsart von JSR ist absolut, dem Befehlswort folgen also LSB und MSB der Sprungadresse. Folglich ist JSR ein 3-Byte-Befehl.

Hingegen ist RTS ein 1-Byte-Befehl, die Adressierungsart ist implizit. Das ist auch recht logisch. Der Rücksprung aus dem Unterprogramm folgt ja immer genau an die Stelle, von der aus das Unterprogramm angesprungen wurde. Mithin muß zu RTS selbst keine Adresse mehr angegeben werden. Das Besondere an einem Unterprogrammaufruf ist ja eben gerade, daß sich der Prozessor die Rücksprungadresse „merkt".

Übersetzt in Assembler könnte unsere Unterprogrammstruktur wie folgt aussehen:



Lassen Sie sich hier nicht verwirren: RTS dient sowohl zum Abschluß des Unterprogramms im Adressbereich von 5500 bis 5530 als auch zur Beendigung des Programms (Adresse 5170). Der Grund dafür ist ganz einfach. Wenn wir bisher ein Assemblerprogramm von BASIC oder von 'T.EX.AS.' aus gestartet haben, so wurde unser Assemblerprogramm in jedem Fall als Unterprogramm aufgerufen – eben als Unterprogramm des BASIC-Interpreters oder als Unterprogramm von 'T.EX.AS.'.

Wir wollen uns jetzt einmal ein konkretes Beispiel für einen Unterprogrammaufruf ansehen. Die Aufgabenstellung lautet wie folgt.

Innerhalb eines umfangreicheren Programms ist es sehr häufig notwendig, daß Inhalte aus dem einen Speicherbereich in einen anderen Bereich verschoben werden müssen. Es soll nun ein Unterprogramm entwickelt werden, welches diese Verschiebung vornimmt und dabei so universell ist, daß von dem aufrufenden Hauptprogramm aus gewählt werden kann, welcher Bereich wohin verschoben werden soll. Zur Vereinfachung wollen wir dabei annehmen, daß maximal 255 Bytes auf einmal geschoben werden müssen.

Um das Programm allgemein verwertbar zu halten, wollen wir mit Labels arbeiten. Dabei treffen wir die folgenden Vereinbarungen:

Label	Adresse	Bedeutung
ANZAHL	10	Anzahl der zu verschiebenden Bytes
VONLSB	11	LSB der VON-Adresse
VONMSB	12	MSB der VON-Adresse
NACHLSB	13	LSB der NACH-Adresse
NACHMSB	14	LSB der NACH-Adresse

Frei festlegbar sind also, wieviele Bytes von welcher Adresse nach welcher Adresse verschoben werden sollen. Achtung: Die Speicherzellen 10 bis 14 werden normalerweise vom BASIC-Interpreter benutzt, so daß unser Programm nicht von BASIC aus aufgerufen werden darf. Sie können auch einen freien Bereich zum Beispiel von 3010 bis 3014 wählen, was allerdings den Nachteil hat, daß die Adressierungsart „Zero-Page" nicht mehr möglich ist.

Unsere Unterprogrammroutine sieht mit den oben angegebenen Adressen wie folgt aus:

```
SCHIEBEN  LDY  10          ;Start des Programms
SCHLEIFE  LDA  (11), Y     ;(11/12 indirekt) plus Y
          STA  (13), Y     ;(13/14 indirekt) plus Y
          DEY              ;Y-Register von Inhalt von 10
          BNE  SCHLEIFE    ;bis 0 zählen, dann Rück-
          RTS              ;sprung zum Hauptprogramm
```

Wir haben bewußt auf die Angabe von Adressen oder Zeilennummern verzichtet. Sie sind mittlerweile weit genug fortgeschritten in der Assemblerprogrammierung, so daß Sie das Programm selbständig in jeden beliebigen Speicherbereich legen können, entweder mit Direkt- oder mit Editor-Assembler.

Ein Beispiel für einen Programmteil, das unser Unterprogramm benutzt (aufruft), wollen wir Ihnen auch nicht vorenthalten:

```
                               ;
AUFRUF    LDA  #ANZAHL   ;Anzahl der zu verschiebenden
          STA     10     ;Bytes in Adresse 10
          LDA  #VONLSB   ;LSB und MSB der VON-Adresse
          STA     11     ;in Adressen 11 und 12
          LDA  #VONMSB
          STA     12
          LDA  #NACHLSB  ;LSB und MSB der NACH-Adresse
          STA     13     ;in Adressen 13 und 14
          LDA  #NACHMSB
          STA     14
          JSR  SCHIEBEN  ;Aufruf der Verschieberoutine
```

Wie Sie sehen, werden zunächst die entsprechenden Speicherzellen mit den gewünschten Werten belegt und anschließend wird das Unterprogramm „SCHIEBEN" aufgerufen. Wenn nun innerhalb des Gesamtprogramms mehrmals eine Verschiebung benötigt wird, so brauchen jeweils nur die Adressen 10 bis 14 mit den aktuellen Werten belegt zu werden, die eigentliche Verschieberoutine hingegen steht nur ein einziges Mal im Programm und wird immer wieder mit „JSR SCHIEBEN" aufgerufen.

Während in dem gezeigten Beispiel die Übergabe von Werten vom Haupt- zum Unterprogramm in Speicherzellen erfolgt, kann dies natürlich bei bis zu nur drei Werten auch mit den internen Prozessorregistern – Akkumulator sowie X- und Y-Register – erfolgen. Dadurch verkürzt sich der aufrufende Programmteil. Allerdings ist dies nicht in allen Fällen möglich – Sie kennen selbst die Einschränkungen der Prozessorregister.

Damit wollen wir das Gebiet „Unterprogrammtechnik" zunächst verlassen. Das sich dahinter verbergende Prinzip der sog. „Stackverarbeitung" werden wir erst im nächsten Kapitel besprechen. Sie wissen jedoch bereits jetzt genug von Unterprogrammen, um diese in Ihren eigenen Programmen einsetzen zu können. Fühlen Sie sich ermutigt, selbständig in diese Richtung weiter zu experimentieren.

Zusammenfassung

Wir haben anhand einiger Beispiele die in dem vorgegangenen Kapitel gelernten neuen Adressierungsarten und Programmiertechniken in der Praxis erprobt.

Um die Assemblerprogrammierung komfortabler zu gestalten, ordnet man Adressen Namen zu. Statt „Name" sagt man allgemein „Label" oder auch „Marke". Sinnfällig gewählte Namen bleiben leichter im Gedächtnis als reine Zahlen. Manche Assemblersysteme verarbeiten Labels, andere nicht.

Unabhängig davon unterscheidet man zwischen Direkt- oder Line-by-Line-Assemblern und Editor-Assemblern. Bei einem Direkt-Assembler werden eingegebene Assembleranweisungen sofort in Assemblercode umgewandelt. Um diesen Code wieder als Text darzustellen, muß der Code zurückgewandelt („disassembliert") werden. Dabei gehen alle Labels verloren. Außerdem können praktisch keine Befehle eingefügt oder gelöscht werden.

Bei einem Editor-Assembler wird der Assemblertext zunächst vollständig in einem Editor geschrieben und steht als Text zur Verfügung („Quelltext"). Die Umwandlung in Assemblercode (auch „Objectcode") ab geschieht in einem zweiten Vorgang („Assemblierung"). Pseudokommandos steuern die Assemblierung. Der Editor erlaubt im allgemeinen ein recht komfortables Arbeiten mit Zeilennummern etc. Allerdings muß das Programm nach jeder Änderung erneut assembliert werden.

Die folgenden Anweisungen sind in diesem Kapitel neu hinzugekommen:

NOP	„no operation", keine Operation
TAX	Transfer Akku nach X-Register
TAY	Transfer Akku nach Y-Register
TXA	Transfer X-Register nach Akku
TYA	Transfer Y-Register nach Akku
TSX	Transfer Stackpointer nach X-Register
TXS	Transfer X-Register nach Stackpointer
JSR	„jump to subroutine", Unterprogrammsprung
RTS	„return from subroutine", Rücksprung aus Unterprogramm

NOP sowie alle Transfer-Anweisungen sind 1-Byte-Befehle mit impliziter Adressierung. Die Transfer-Befehle übertragen eine Kopie des Inhaltes des erstgenannten Registers in das zweitgenannte.

JSR ist ein 3-Byte-Befehl. Dem Befehlswort folgt die Sprungadresse als LSB und MSB. JSR bewirkt – genau wie JMP – einen Sprung zur angegebenen Adresse. Im Unterschied zu JMP merkt sich der Prozessor bei JSR, von wo aus der Sprung erfolgt. Bei RTS kommt es zu einem Rücksprung aus dem Unterprogramm an eben die Stelle, von wo aus der Hinsprung erfolgte – genauer, an den nachfolgenden Befehl. RTS ist ein 1-Byte-Befehl mit der Adressierungsart implizit.

Mit JSR und RTS lassen sich alle – zum Beispiel von BASIC – bekannten Unterprogrammtechniken ausnutzen.

Kapitel 9: Bitoperationen, Stack, Prozessoraufbau, Interrupt
- AND, ORA, EOR, BIT, ASL, LSR, ROL, ROR, PHA, PLA, PHP, PLP, BRK, RTI.

Wir wollen in diesem Kapitel einige wesentliche theoretische Grundlagen der Programmierung in Assembler besprechen. Dabei werden wir zunächst noch einmal in den Bereich „Bits und Bytes" einsteigen, dann die Stackkonzeption – bei JSR und RTS schon angesprochen – genau erläutern und dabei gleichzeitig den logischen Gesamtaufbau des Mikroprozessors skizzieren. Schließlich werden wir das ebenfalls schon kurz angesprochene Thema „Interrupt" vertiefen. Dabei werden wir nach und nach die noch verbliebenen uns bisher unbekannten Assemblerbefehle kennenlernen.

Nach erfolgreichem Abschluß dieses Kapitels werden Sie alle wesentlichen Prinzipien der Assemblerprogrammierung kennen – wenn auch vielleicht noch nicht beherrschen. Aber dies ist schließlich nur eine Frage der Erfahrung, die Sie im Laufe der Zeit noch sammeln werden.

Beginnen wir mit Bitoperationen. Der Begriff „Bit" ist Ihnen schon lange geläufig und Sie wissen auch, daß acht Bits ein Byte bilden. Nun haben wir bisher in praktisch allen Fällen ein Byte als Einheit betrachtet, ohne die acht Bits speziell zu berücksichtigen. Wenn wir zum Beispiel die Anweisung ...

```
LDA #240
```

... verwenden, so interessiert uns der Wert 240 als solches, jedoch nicht die interne Bitkombination. Sie wissen aber, letztendlich rechnen wir immer mit einzelnen Bits, auch wenn wir davon oft nicht direkt etwas merken.

Der einzige Fall, in dem wir bisher die acht Bits eines Bytes einzeln betrachteten, ist das Statusregister des Prozessors. Dort haben die einzelnen Bits sogar Namen wie Carry-Bit oder Carry-Flag etc.

Nun gibt es aber auch Anwendungen, bei denen den einzelnen Bits eines „ganz normalen" Byte – also nicht nur des Statusregisters – besondere, voneinander unabhängige Bedeutungen zukommen. Ein einfaches Beispiel: Die acht Bits steuern über elektrische Leitungen, die vom Computer nach außen geführt sind, verschiedene Geräte. Im einfachsten Fall bedeutet eine 0, daß das betreffende Elektro-Gerät aus- und eine 1, daß es eingeschaltet ist. Wenn wir unterstellen, daß alle Geräte voneinander unabhängig gehandhabt werden sollen, dann müssen wir Möglichkeiten finden, einzelne Bits zu setzen und zu löschen.

Auch die Umwandlung eines Codes in einen anderen kann zum Beispiel durch die Manipulation einzelner Bits geschehen. Wenn Sie das Gebiet erst einmal kennen, werden Sie feststellen, daß es vielfältige Möglichkeiten für Bitoperationen gibt.

Wenn Sie mathematisch belastet sind, kennen Sie vielleicht schon die sog. „Boole'sche Operationen", benannt nach dem Mathematiker George Boole. „Boole'sche Operationen" sind solche mit Binärzahlen (auch Dualzahlen). Die wichtigsten „Boole'schen Operationen" heißen AND („und"), OR („oder") und NOT („nicht"), erweiterte Operationen sind „Exclusive OR" und andere.

Die dafür gültigen Gesetze sind recht einfach zu verstehen. Wir haben sie hier aufgelistet (jeweils auf ein einziges Bit bezogen). Mit AND und OR werden zwei Bits nach bestimmten Regeln miteinander verknüpft, wobei ein Ergebnis entsteht.

AND (UND)	OR (ODER)
0 AND 0 = 0	0 OR 0 = 0
0 AND 1 = 0	0 OR 1 = 1
1 AND 0 = 0	1 OR 0 = 1
1 AND 1 = 1	1 OR 1 = 1

Nehmen wir ein Beispiel aus der Tabelle: Wenn eine 0 und eine 1 mit AND verknüpft werden, so ist das Ergebnis 0, werden sie mit OR verknüpft, ist das Ergebnis 1. Die dahintersteckende Logik:

Das Ergebnis einer Verknüpfung mit AND (übersetzt „und") ist nur dann 1, wenn das erste Bit UND das zweite Bit auf 1 sind. Andernfalls ist das Ergebnis gleich 0.

Das Ergebnis einer Verknüpfung mit OR (übersetzt „oder") ist dann 1, wenn das erste Bit ODER das zweite Bit (oder auch beide) 1 sind. Andernfalls ist das Ergebnis gleich 0.

Während bei AND und OR immer jeweils zwei Bits miteinander in Beziehung gebracht – verknüpft – werden, bezieht sich die Operation NOT nur auf ein einziges Bit. Die Regeln muten geradezu simpel an:

NOT 0 = 1
NOT 1 = 0

Die Operation NOT (übersetzt „nicht") wird auch sehr häufig „Negation" genannt. Es gibt allerdings keinen Assemblerbefehl für NOT.

Was hat es nun mit der „Spielerei" mit AND, OR und NOT auf sich? Nun, für die Zahlen unseres alltäglichen Lebens kennen wir eine Vielzahl von arithmetischen Operationen, die vier Grundrechenarten sind sicherlich die bekanntesten. Bei der Addition zum Beispiel werden zwei Zahlen miteinander in Beziehung gesetzt, so daß eine dritte Zahl als Ergebnis dabei herauskommt. Die Regeln der Addition lernen wir schon in einem so frühen Alter, daß wir sie wie selbstverständlich beherrschen.

Für das „Rechnen" mit den Binärzahlen existieren spezielle Operationen wie AND, OR und NOT, die den Besonderheiten des Binärsystems angepaßt sind. Übrigens gibt es noch weitere Binäroperationen. Eine davon wollen wir noch kennenlernen, da sie auch für die Assemblerprogrammierung von Bedeutung ist. Sie heißt „EOR", was als Abkürzung steht für „exclusive or", zu deutsch „Exklusiv-Oder".

Hier die Verknüpfungsregeln für EOR:

EOR (Exklusiv-ODER)

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

In Worten: Das Ergebnis einer Verknüpfung mit EOR ist dann 1, wenn allein (EXKLUSIV) das erste Bit ODER allein (EXKLUSIV) das zweite Bit 1 ist. Vergleichen Sie dies mit der OR-Operation, um sich den Unterschied klar zu machen.
Man kann auch sagen, das Ergebnis von EOR ist dann 1, wenn beide Bits unterschiedlich (0 bezw. 1) sind.

Wir wollen gleich das Gelernte in der Praxis anwenden und einmal zwei Bytes mit AND verknüpfen. Dazu werden jeweils die entsprechenden Bits aus dem einen und dem anderen Byte gemäß der Definition von AND behandelt.

```
1. Byte:     1 0 1 1 0 1 0 0
2. Byte:     0 1 1 0 1 1 1 0     AND
             ---------------
Ergebnis:    0 0 1 0 0 1 0 0
```

Sie sehen, entsprechend der Festlegung von AND werden im Ergebnisbyte nur diejenigen Bits auf 1 gesetzt, bei denen im ersten UND im zweiten Byte das Bit gesetzt ist.

Mit OR verknüpft sieht unser Schema wie folgt aus:

```
1. Byte:     1 0 1 1 0 1 0 0
2. Byte:     0 1 1 0 1 1 1 0     OR
             ---------------
Ergebnis:    1 1 1 1 1 1 1 0
```

Im Ergebnisbyte sind alle Bits auf 1 gesetzt, bei denen im ersten ODER im zweiten Byte (oder in beiden!) ein Bit gesetzt ist. Sie sehen, wenn man nur die Regeln beherrscht, ist das Rechnen mit AND und OR einfach, im Grunde einfacher noch als Addieren oder Subtrahieren.

Führen wir auch mit EOR die Verknüpfung unserer beiden Bytes durch:

1. Byte:	1 0 1 1 0 1 0 0	
2. Byte:	0 1 1 0 1 1 1 0	EOR

Ergebnis:	1 1 0 1 1 0 1 0	

Im Ergebnisbyte ist nur dann ein Bit gesetzt, wenn nur im ersten oder nur im zweiten Byte an derselben Stelle ebenfalls ein Bit gesetzt ist.

Nachdem Sie einige der mathematischen Grundlagen der „Boole'schen Operationen" kennengelernt haben, wollen wir uns jetzt ansehen, wie so etwas in Assembler programmiert wird. Für AND, OR und EOR existieren auch tatsächlich Assemblerbefehle. Die Befehlsworte heißen „AND", „ORA" und „EOR". Da OR nur aus zwei Buchstaben besteht und andererseits jedes Assemblerbefehlswort genau drei Buchstaben lang sein muß, hat man „OR" erweitert zu „ORA" für „OR Accumulator".

Kommen wir nun auf unser anfängliches Beispiel von den acht verschiedenen Geräten, die von acht Bits gesteuert wreden, zurück. Wir legen die Zusammengehörigkeit auf die folgende höchst einfache Weise fest:

Gerätenumerierung	7	6	5	4	3	2	1	0
Bit-Numerierung	7	6	5	4	3	2	1	0
Inhalt des Byte	0	0	0	0	0	0	0	0

Als Computerfachleute beginnen wir grundsätzlich alle Dinge von 0 an zu zählen, die Bits werden also von 0 bis 7 statt von 1 bis 8 durchnumeriert. Das ist recht sinnvoll, es entspricht nämlich exakt der Wertigkeit der einzelnen Bits (s. Kapitel 3). Zu Anfang sind alle Bits auf 0 gesetzt und somit alle acht angeschlossenen Geräte ausgeschaltet.

Nun wollen wir uns überlegen, wie wir die Geräte Nummer 0, 2 und 7 anschalten können, d.h., wie wir am einfachsten die entsprechenden Bits auf 1 setzen können. Zur Erhöhung der Übersichtlichkeit wollen wir dem Byte, welches zur Steuerung unserer Geräte dient, einen Namen – Label – geben, sagen wir OUTPUT. „Output" heißt „Ausgabe" (als das Gegenteil von „Eingabe", nicht „Einnahme"!), und wir wollen ja hier Steuersignale an unsere acht Geräte ausgeben.

Setzen wir also die Bits 0, 2 und 7. Unser Byte sieht dann so aus '10000101', oder dezimal 133 (die Umwandlung einer Binär- in eine Dezimalzahl wird auf Seite 3-7 erklärt). Sehen Sie ruhig nochmals in Kapitel 3 nach.

Das kurze Programmstück in Assembler, das die drei genannten Geräte einschaltet, könnte zum Beispiel wie folgt aussehen:

```
LDA  #133
STA  OUTPUT
```

Soweit ist das nichts Neues für Sie. Jetzt kommen wir aber zu einer etwas schwierigeren Aufgabe: Das Gerät Nummer 5 soll nun eingeschaltet werden, ohne daß davon die übrigen Geräte berührt werden. Mit anderen Worten: Bit 5 muß gesetzt werden, ohne daß die anderen Bits beeinflußt werden. Mit der OR-Operation kann dieses Problem leicht gelöst werden.

```
1. Byte (unbekannt):       ? ? ? ? ? ? ? ?
2. Byte (setzt Bit 5)      0 0 1 0 0 0 0 0        OR
                           ---------------
Ergebnis:                  ? ? 1 ? ? ? ? ?
```

Bit 5 des Ergebnisbyte wird also in jedem Fall gesetzt, die anderen Bits des Ergebnisses entsprechen exakt denen des ersten Bytes. '00100000' entspricht der Dezimalzahl 32 (2 hoch 5).

In Assembler sieht das zum Beispiel so aus (Es gibt noch eine Reihe von anderen Möglichkeiten, dies zu programmieren.):

```
LDA  OUTPUT
ORA  #32
STA  OUTPUT
```

Die acht OUTPUT-Bits werden in den Akkumulator geladen und mit dem Wert 32 „geORet". Anschließend wir die geänderte Bit-Kombination – jetzt mit gesetztem Bit 5 – in OUTPUT zurückgespeichert.

Nehmen wir nun eine Aufgabe, die mit AND zu lösen ist. Die Geräte 5 und 6 sollen ausgeschaltet werden, ohne daß die anderen Geräte verändert werden. Dazu nehmen wir einfach die folgende AND-Verknüpfung:

```
1. Byte (unbekannt):           ? ? ? ? ? ? ? ?
2. Byte (löscht Bit 5 und 6)   0 0 0 1 1 1 1 1      AND
                               ---------------
Ergebnis:                      ? 0 0 ? ? ? ? ?
```

Die zu löschenden Bits 5 und 6 werden mit 0 „geANDet". Das Dezimaläquivalent zu '10011111' lautet 159 (1+2+4+8+16+128).

In Assembler ausgedrückt:

```
LDA  OUTPUT
AND  #159
STA  OUTPUT
```

Allen Beispielen ist gemeinsam, daß einzelne Bits gezielt gesetzt oder gelöscht werden, ohne daß davon die übrigen Bits verändert werden. Deshalb ist auch der Zustand der anderen Bits (gesetzt oder nicht) unerheblich.

Jetzt wollen wir einmal alle acht Bits unabhängig voneinander verändern. Die Aufgabe lautet diesmal: Jedes einzelne Gerät soll umgeschaltet werden. Ein eingeschaltetes Gerät soll also ausgeschaltet werden und umgekehrt. Sie als angehender Profi nehmen dafür natürlich sofort EOR:

```
1. Byte (unbekannt):          ? ? ? ? ? ? ? ?
2. Byte (negiert alle Bits):  1 1 1 1 1 1 1 1      EOR
                              ----------------------------
Ergebnis:                     ? ? ? ? ? ? ? ?
```

Ein gesetztes Bit des ersten Bytes erscheint im Ergebnis als 0, ein gelöschtes Bit hingegen als 1. Ein EOR mit allen acht Bits gesetzt entspricht also der schon angesprochnen NOT-Funktion. Aus diesem Grund existiert in Assembler auch keine eigene NOT-Operation, sie läßt sich ja mit EOR einfach nachbilden.

Und EOR gibt es natürlich in Assembler:

```
LDA  OUTPUT
EOR  #255
STA  OUTPUT
```

Wenn alle acht Bits gesetzt sind, so entspricht das ja der Dezimalzahl 255.

Wir haben bisher AND und ORA sowie EOR ausschließlich mit der Adressierungsart „direkt" kennengelernt. Alle drei Befehle erlauben weiterhin die Adressierungen Zero-Page, Zero-Page X-indiziert, absolut, absolut X- und Y-indiziert, indiziert-indirekt und indirekt-indiziert. Damit sind die Möglichkeiten recht vielfältig. In jedem Fall wird der Akkumulatorinhalt entsprechend der Definitionen der Boole'schen Operationen mit dem Inhalt der gewählten Speicherzelle bzw. direkt verknüpft. Das Ergebnis steht danach im Akkumulator.

AND, ORA und EOR sind aber nicht die einzigen bit-bezogenen Befehle in Assembler. Es gibt noch ASL, LSR, ROL und ROR, die Bit-Verschiebungen innerhalb eines Bytes erlauben.

Wir unterscheiden zwischen den Operationen „Verschieben" („shift") und „Rotieren" („rotate"). In beiden Fällen werden innerhalb eines Registers – zum Beispiel des Akkumulators – alle Bits um eine Position nach rechts oder links gebracht. Bei „Verschieben" wird an der freien Position eine Null nachgeschoben, bei „Rotieren" rotieren die Bits im „Kreis", d. h., was rechts herausgeschoben wird, kommt von links wieder herein und umgekehrt. In jedem Fall wird das Carry-Bit als eine Art neuntes Bit behandelt.

Wenn Ihnen das alles noch ein bißchen mysteriös vorkommt, dann klärt das folgende Schaubild bestimmt die Unklarheiten. „BO", „B1" usw. steht für „Bit 0", „Bit 1" usw., „C" bedeutet „Carry-Bit", „0" ist der Wert Null.

Rechts-Verschiebung (LSR, „Logic shift right")

vorher:	B7	B6	B5	B4	B3	B2	B1	B0	C
nachher:	0	B7	B6	B5	B4	B3	B2	B1	B0

Links-Verschiebung (ASL, „arithmetic shift left")

vorher:	B7	B6	B5	B4	B3	B2	B1	B0	C
nachher:	B6	B5	B4	B3	B2	B1	B0	0	B7

Rechts-Rotation (ROR, „rotate right")

vorher:	B7	B6	B5	B4	B3	B2	B1	B0	C
nachher:	C	B7	B6	B5	B4	B3	B2	B1	B0

Links-Rotation (ROL, „rotate left")

vorher:	B7	B6	B5	B4	B3	B2	B1	B0	C
nachher:	B6	B5	B4	B3	B2	B1	B0	C	B7

Im Unterschied zu den Boole'schen Operationen AND, ORA und EOR beeinflussen die Verschiebe- und Rotier-Anweisungen also nicht nur einzelne Bits, sondern schon alle acht Bits und sogar das Carry-Bit. Dennoch stehen auch hier die einzelnen Bits im Vordergrund, die Zahl als solche interessiert nicht – im Gegensatz zum Beispiel zu ADC, wo uns das Ergebnis der Addition interessiert und nicht die interne Bit-Kombination.

Mögliche Adressierungsarten für ASL, LSR, ROL und ROR sind: Akkumulator, Zero-Page X-indiziert, absolut und absolut X-indiziert. Achtung: Das Ergebnis der Operation steht immer in dem Register oder der Speicherzelle, innerhalb dem bzw. der geschoben oder rotiert wird. Der Akkumulator ist das nur bei der Adressierungsart „Akkumulator", sonst ist es die entsprechende Speicherzelle.

So wird bei diesem Beispiel die Speicherzelle 6230 verändert, der Akkumulator bleibt völlig unbeeinflußt:

```
ROR 6230             ;Rechtsrotation Adresse 6230
```

Hingegen werden in diesem Beispiel alle Bits des Akkumulatorinhalts um zwei Positionen nach links verschoben:

```
ASL                  ;zweimalige Lingsverschiebung
ASL                  ;innerhalb des Akkumulators
```

Die Verschiebe- und Rotations-Befehle können also je nach Adressierungsart 1-, 2- oder 3-Byte-Befehle sein.

Das Prozessorstatusregister wird wie folgt beeinflußt: /- und N-Flag werden entsprechend dem Ergebnis der Verschiebung bzw. Rotation gesetzt oder gelöscht, das V-Flag bleibt unverändert und das Carry-Flag wird – wie bereits erklärt – als neuntes Bit der Verschiebung bzw. Rotation berücksichtigt.

Im folgenden zugegebenermaßen reichlich sinnlosen Beispiel wird ein Bit von rechts nach links bis zum Carry-Flag und wieder zurückgeschoben, dann beginnt das Spiel von vorne.

```
         LDA #1        ;Bit 0 des Akku setzen, restli-
         CLC           ;che Bits und Carry-Bit löschen
LINKS    ROL           ;Linksrotation, bis die 1 im
         BCC LINKS     ;Carry-Bit steht
RECHTS   ROR           ;Rechtsrotation, bis die 1 im
         BCC RECHTS    ;Carry-Bit steht
         JMP LINKS     ;Und wieder von vorne
```

(Sie als erfahrener Programmierer wissen natürlich, daß wir ein Byte Speicherplatz einsparen können, wenn wir „BCS LINKS" nehmen statt „JMP LINKS". BCS bewirkt an dieser Stelle auf jeden Falle einen Sprung, denn wenn die BCC-Bedingung nicht erfüllt ist, muß ja die – umgekehrte – BCS-Bedingung erfüllt sein).

Nachdem Sie nun sieben verschiedene Assemblerbefehle kennengelernt haben, die die Veränderung eines Bytes auf Bit-Ebene ermöglichen, wollen wir zum Abschluß dieses Themas noch diejenige Assembleranweisung besprechen, die Bit-Vergleiche erlaubt. Sie wissen, üblicherweise nehmen wir für Vergleiche in Assembler CMP und CPX oder CPY. Diese sind jedoch weitgehend nur für Zahlen geeignet, jedoch nicht, wenn es um einzelne Bits geht. Der dafür richtige Befehl heißt bezeichnenderweise „BIT". BIT führt ein AND zwischen einer Speicherstelle und dem Akkumulator aus, speichert aber das Ergebnis der Verknüpfung nicht im Akkumulator. Vielmehr geht das Ergebnis selbst verloren, aber das Statusregister wird entsprechend dem Ergebnis gesteuert.

BIT beeinflußt die Statusflags Zero (Z), Negativ (N) und Overflow (V). Die möglichen Adressierungsarten sind Zero-Page und absolut. Wie auf jede Vergleichsanweisung folgt auf BIT praktisch immer ein Branch-Befehl.

Ein Beispiel: Ein externes Gerät (zum Beispiel eine Tastatur) ist an eine Speicherzelle – wir wollen sie INPUT („Eingabe") nennen – angeschlossen. Wir wollen den kurzen Teil eines Programms schreiben, der darauf wartet, bis eine bestimmte Taste gedrückt wird. Bei dem Tastendruck ändert sich die Speicherstelle INPUT derart, das Bit 2 auf 1 gesetzt wird, unabhängig von den restlichen Bits. Gewartet wird somit auf das Bitmuster '?????1??'. Da uns die anderen Bits nicht interessieren, müssen wir sie für die AND-Verknüpfung auf 0 setzen (BIT beinhaltet AND, ohne das Ergebnis aufzuheben). Das ergibt eine Bit-Kombination '00000100', dezimal 4. Das kann in Assembler so aussehen:

```
          LDA  #4
WARTEN    BIT INPUT
          BEQ WARTEN
```

Das Programm verbleibt in der Warteschleife, bis Bit 2 der Speicherstelle INPUT auf 1 gesetzt wird (Tastendruck).

So, damit wollen wir den Bereich „Bitoperationen" abschließen. Es ist gar nicht wichtig, daß Sie sich jetzt schon die besprochenen Möglichkeiten im Detail merken, behalten Sie aber im Hinterkopf, daß der Prozessor solche Befehle kennt. Im Laufe Ihrer Programmierpraxis werden Ihnen noch früh genug Probleme begegnen, die Sie dann mit den hier besprochenen Anweisungen elegant lösen können.

Kommen wir nun zu einem fast noch wichtigeren Gebiet der Assemblerprogrammierung, den „Stackoperationen". Dazu müssen wir zunächst den Begriff „Stack" klären. Die deutsche Übersetzung dafür lautet „Stapel", auch „Ablagespeicher".

Nehmen wir an, Sie haben eine Reihe von Büchern, die Sie aufeinander stapeln möchten. Dann legen Sie zuerst das erste Buch hin, darauf das zweite, wiederum darauf das dritte usw. Der Stapel wird so hoch, wie Sie Bücher haben. Um nun ein Buch wieder aus dem Stapel herausnehmen zu können, müssen Sie alle darauf liegenden Bücher entfernen. Sie nehmen das oberste Buch und legen es weg, dann das nächstoberste usw. bis Sie an das Buch gekommen sind, das Sie haben wollten. Das Buch, das Sie zuallererst auf den Stapel gelegt haben – also das unterste – ist das Buch, das Sie beim Abbau des Stapels als letztes wieder herausbekommen. Das Buch, das Sie als leztes auf den Stapel gelegt haben hingegen ist dasjenige, das Sie als erstes wieder von dem Stapel nehmen können. Man nennt dieses Prinzip „LIFO", „last in – first out", zu deutsch „zuletzt hinein – zuerst heraus".

Im Prinzip genau wie dieser Bücherstapel funktioniert der Stack des Computers. Nur daß es sich hierbei natürlich nicht um Bücher handelt, sondern um Speicherstellen, die mit Werten – statt Büchern – gefüllt werden. Bei dem Prozessor 6502 und Abarten davon liegt der Stackbereich von Adresse 256 bis 511, auch genannt „Page 1". Die Bezeichnung „Page 1" rührt daher, daß in diesem Adressbereich bei einer Unterteilung in LSB und MSB das MSB immer 1 ist (s. auch Seite 7-2). Dabei entspricht Adresse 511 der untersten Position des Bücherstapels und 256 der obersten (wichtig!). Es können also nicht mehr als 256 (511 – 256) Werte auf einmal gestapelt werden.

Obwohl man bei dem angenommenen Bücherstapel noch sehen kann, wieviele Bücher da sind, wird es bei einer großen Anzahl von Büchern dennoch sinnvoll sein, einen Zähler zu haben. Dieser Zähler erhöht sich um eins, wenn ein Buch auf den Stapel gelegt wird und erniedrigt sich um eins, wenn ein Buch weggenommen wird.
Da man im Computerstack natürlicherweise nicht sehen kann, wieviele Werte gestapelt sind, ist hier ein Zähler unbedingt erforderlich. Wir nennen diesen Zähler „Stackpointer", übersetzt „Stapelzeiger". Der Stackpointer zeigt immer auf genau die Adresse, die den nächsten freien Wert des Stacks enthält.

Wozu wird nun aber der Stack gebraucht? Bei allen bisherigen Adressierungen waren wir davon ausgegangen, daß entweder die genaue Stelle oder zumindest eine Beziehung zu der genauen Stelle einer Speicheradresse bekannt ist. Es gibt aber Programmstrukturen, bei denen nicht die genaue Stelle von Daten angegeben werden kann, sondern lediglich die Reihenfolge der Daten. Ein Beispiel dafür ist die Unterprogrammtechnik.

Wie Sie bereits im letzten Kapitel erfahren haben, erfolgt ein Unterprogramm mit JSR, die Rückkehr mit RTS. Hauptmerkmal dabei ist, daß sich der Prozessor bei dem JSR-Sprung merkt, von welcher Adresse aus gesprungen wird und bei RTS automatisch mit der nächstfolgenden Adresse des Hauptprogramms weiterarbeitet.

```
8000  JSR  56000
8003  LDA   #255
```

In diesem Beispiel wird die Programmausführung zu Adresse 56000 übergeben und verbleibt dort, bis ein RTS-Befehl auftritt. Danach setzt die Programmbearbeitung bei Adresse 8003 fort. Der Prozessor hat sich bei JSR „gemerkt", daß dies von Adresse 8000 aus geschieht und weiß – da JSR immer ein 3-Byte-Befehl ist – daß der nachfolgende Befehl bei Adresse 8003 beginnt.

Und dieses „Merken" geschieht, indem der Prozessor die Rücksprungadresse – also im Beispiel 8003 – als LSB und MSB auf dem Stack ablegt. Gleichzeitig wird noch der Stackpointer verändert.

Nehmen wir einmal an, daß der Stackpointer zu Anfang unseres Beispiels auf Adresse 511 zeigt. Machen Sie sich klar: Das ist zwar die höchste Adresse, sie entspricht aber der untersten Position unseres Bücherstapels. Mit anderen Worten: Es liegt noch kein einziges Buch bisher auf dem Stapel.

Während nun die Anweisung 'JSR 56000' ausgeführt wird, wird die Rücksprungadresse – genauer gesagt die Rücksprungadresse minus 1 – auf dem Stack abgelegt. Wenn der Stackpointer vorher auf Adresse 511 im Stack gezeigt hat, so werden also jetzt die Adressen 511 und 510 mit MSB und LSB (in dieser Reihenfolge!) belegt und der Stackpointer wird auf Adresse 509 gesetzt.

Die Inhalte von 511 und 510 sind MSB und LSB der Rücksprungadresse minus 1 (über diese „minus 1" sprechen wir gleich noch). In unserem Beispiel wäre das die Adresse 8002 (Rücksprungadresse ist 8003, minus 1), geteilt in 66 (LSB) und 31 (MSB). Sie erinnern sich an die Rechenweise: 31 mal 256 plus 66 ist gleich 8002.

Sie können sich diese Zusammenhänge an folgender Übersicht deutlich machen:

Adresse	Inhalt	Stack		Stackpointer
8000	JSR	511	31 (MSB)	511
8001	192 (LSB)	510	66 (LSB)	510
8002	218 (MSB)	509	--------	509
8003	LDA			

Die geringere Adresse beinhaltet immer das LSB, die höhere das MSB (auch innerhalb des Stacks).

Die auf dem Stack als LSB und MSB gespeicherte Adresse ist die letzte Adresse der JSR-Anweisung (8002). Der nächste Befehl beginnt aber genau eine Adresse höher (8003). Auf dem Stack steht somit nicht die korrekte Rücksprungadresse, sondern die Rücksprungadresse minus 1. Dies wird korrigiert beim Rücksprung selbst, der da bei RTS stattfindet. RTS macht also aus der 8002 die gewünschte 8003, so daß – in unserem Beispiel – die Programmsequenz mit LDA... fortgesetzt wird.

Wenn innerhalb eines Unterprogramms ein weiteres Unterprogramm aufgerufen wird usw. so füllt sich der Stack nach und nach auf:

Stackadresse	Inhalt
511	MSB1
510	LSB1
509	MSB2
508	LSB2
507	MSB3
506	LSB3

Wie wir besprochen haben, dient der Stack also einmal zur Abspeicherung der Rücksprungadressen bei JSR-Sprungbefehlen. In diesem Fall wird der Stack quasi „automatisch" verwaltet. Da für jeden JSR-Befehl zwei Adressen (LSB und MSB) benötigt werden, können somit bis zu 128 Unterprogramme ineinander geschachtelt werden. Dann ist der Stack voll. In den meisten BASIC-Versionen sind nur bis zu einigen 20 Unterprogrammebenen erlaubt.

Außer der Stacksteuerung mit JSR und RTS gibt es auch noch Befehle, die direkt einen Wert auf dem Stack ablegen bzw. einen Wert vom Stack herunternehmen. Sie heißen PHA („push accumulator on stack", „lege Akkumulator auf Stack") und PLA („pull accumulator from stack", etwa „hole Akkumulator von Stack"). PHA und PLA sind beide 1-Byte-Befehle mit der Adressierungsart „impliziert". Beide Befehle erniedrigen bzw. erhöhen den Stackpointer jeweils um eins.

PHA überträgt den Akkumulatorinhalt in die nächste Speicherzelle des Stacks und vermindert den Stackpointer (Sie erinnern sich, je weniger Werte auf dem Stack liegen, desto höher zeigt der Stackpointer!). PHA hat keinen Einfluß auf das Statusregister.

PLA erhöht den Stackpointer um eins und überträgt den zuletzt auf den Stack gelegten Wert in den Akkumulator. N- und Z-Flag werden beeinflußt, die anderen Flags nicht.

PHA und PLA werden häufig benutzt zur Übergabe von Daten zwischen einzelnen Programmteilen. Das kann nach folgendem Prinzip ablaufen:

```
5100 LDA  #240          ;Die Werte 240 und 220 werden auf
5103 PHA                ;den Stack gelegt, der Stackpointer
5104 LDA  #220          ;wird entsprechend heruntergesetzt.
5107 PHA                ;Dann wird ein anderes Teilprogramm
5108 JMP  5500          ;ab Adresse 5500 angesprungen.
5111 ...
  .
5170 RTS                ;Programmende (hier unerheblich)
  .
5500 PLA                ;In diesem Programmteil werden 220
5501 TAX                ;und 240 wieder von Stack geholt und
5502 PLA                ;in das X-Register (220) bzw. den
  .                     Akkumulator (240) zur weiteren
  .                     ;Bearbeitung gebracht. Dann Rück-
5530 JMP  5111          ;sprung zu Adresse 5111.
```

Hier sollen zwei Werte 240 und 220 von einem Programmteil an einen anderen übergeben werden. Da – zum Beispiel – keine eigenen Speicherzellen zur Datenspeicherung frei sind, werden die zwei Werte von dem einen Teil auf den Stack gelegt und vom anderen Teil wieder heruntergeholt. Beachten Sie, daß dieses Prinzip nicht bei Unterprogrammen funktioniert, da JSR ja auch den Stack belegt.

Der Stack wird – außer zur Programmierung von Unterprogrammstrukturen – häufig benutzt zur „Kommunikation" zwischen verschiedenen Routinen eines Gesamtprogramms. Werte werden von einer Routine auf den Stack gelegt und von einer anderen heruntergenommen und weiterverarbeitet. Damit das reibungslos funktioniert, müssen Sie als der Programmierer ganz besonders darauf achten, daß auch genau so viele Werte vom Stack wieder genommen werden., wie zuvor darauf abgelegt wurden. Anderenfalls kommt es zu einem „Datenchaos".

Wir haben bisher stets angenommen, daß der Stackpointer bei „leerem" Stack auf Adresse 511 zeigt und sich entsprechend der Befehle JSR, RTS, PHA und PLA erniedrigt bzw. erhöht. Wenn der Computer eingeschaltet wird, so ist der Stackpointer aber keineswegs von Anbeginn an richtig gesetzt. Vielmehr muß dies durch den Programmierer geschehen.
Es gibt in Assembler zwei Befehle zum direkten Datenaustausch mit dem Stackpointer, beide beziehen sich – neben dem Stackpointer – auf das X-Register. Die Befehlsworte sind „TXS" und „TSX". Sie erinnern sich vielleicht – wir haben diese beiden Anweisungen bereits bei den Transfer-Befehlen im vorangegangenen Kapitel angesprochen. „TXS" steht dabei für „transfer X to stackpointer", „transferiere X-Register nach Stackpointer", „TSX" heißt „transfer stackpointer to X", „transferiere Stackpointer nach X-Register". Transferiert wird nur eine Kopie des jeweiligen Wertes, im Ursprungsregister verbleibt dieser also nach wie vor. Der Vergleich mit BASIC muß entfallen, da es in BASIC keinen Stack gibt, die Variable S kann keinesfalls als Ersatz für den Stackpointer genommen werden.
TXS und TSX sind 1-Byte-Befehle mit der Adressierungsart „implizit". TXS macht den Stackpointer inhaltsgleich mit dem X-Register, Flags werden nicht beeinflußt. TSX überträgt eine Kopie des Inhalts des Stackpointers in das X-Register, Z- und N-Flag werden entsprechend gesetzt oder gelöscht.

Eine der ersten Befehlsfolgen in einem Assemblerprogramm, das den Stack nutzt, wird die folgende sein, um den Stackpointer zu „initialisieren":

```
LDX #255          ;setzt den Stackpointer auf
TXS               ;Adresse 511 (Page Eins + 255)
```

Diese „Initialisierung" kann und muß entfallen, wenn Sie ein Assemblerprogramm von BASIC oder 'T.EX.AS.' aus aufrufen, da in diesem Fall der Stackpointer bereits eingerichtet ist.

Bedenken Sie: der Stack ist so gut ausgelegt, daß er immer in der sog. Page 1 liegt, das sind die Adressen von 256 bis 511 einschließlich. Da der Stackpointer selbst nur aus einem Byte besteht, kann er aber nur Zahlen von 0 bis 255 beinhalten. Deswegen wird vom Mikroprozessor automatisch dafür gesorgt, daß der Stackpointer in Page 1 zeigt. Der Stackpointer stellt also praktisch das LSB einer Zwei-Byte-Adresse dar, das MSB ist mit 1 fest vorgegeben. Mit anderen Worten: Eine 0 im Stackpointer zeigt auf Adresse 256, eine 1 auf Adresse 257 usw. bis die 255 im Stackpointer schließlich auf Adresse 511 im Speicher zeigt.

Das folgende Schaubild verdeutlicht das noch einmal:

Speicheradresse				Stackpointer
MSB	LSB	gesamt (256*MSB+LSB)	Page	
0	0	0	0	-
0	1	1	0	-
0	.	.	0	-
0	255	255	0	-
1	0	256	1	0
1	1	257	1	1
1	2	258	1	2
1	.	.	1	.
1	255	511	1	255
2	0	512	2	-
2	1	513	2	-
2	.	.	2	-
.	.	.	.	-

Sie sehen, der Stack liegt ausschließlich in Page 1.

Wir haben gelernt, den Stack mit JSR/RTS für Unterprogramme „automatisch" zu benutzen, können mit PHA/PLA über den Akkumulator einzelne Werte auf den Stack legen und und zurückholen und haben gesehen, wie über das X-Register mit TXS/TSX direkt auf den Stackpointer zugegriffen werden kann. Häufig ist es auch notwendig, den Inhalt des Prozessorstatusregisters auf dem Stack abzulegen. Das Prozessorstatusregister beinhaltet die uns bekannten Statusflags wie Carry, Negativ, Zero usw. Insbesondere bei der sog. „Interrupt-Verarbeitung" spielt die Ablage des Statusregisters auf dem Stack eine sehr elementare Rolle. Wir werden noch darüber sprechen.

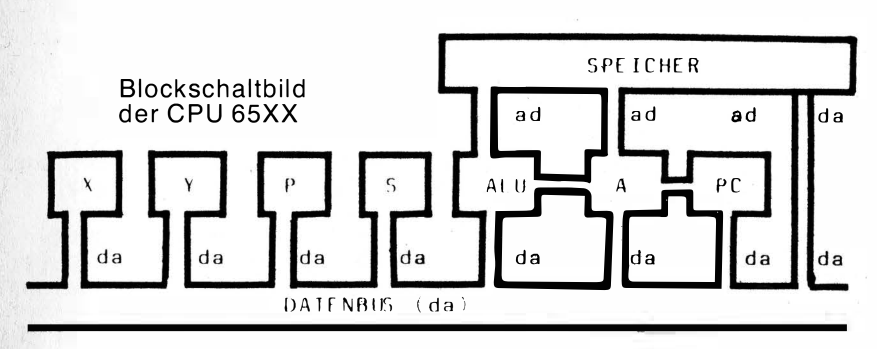
Jetzt wollen wir zunächst die Befehle zur Ablage des Statusregisters auf dem Stack kennenlernen. Die Befehlsworte lauten „PHP", als Abkürzung für „push processor status on stack", „lege Prozessorstatus auf Stack" und „PLP", „pull processor status from stack", „hole Prozessorstatus von Stack". Die Bedeutungen sind im Grunde PHA und PLA recht ähnlich, nur daß hier natürlich nicht der Akkumulator, sondern eben das Statusregister angesprochen wird.

PHP überträgt den Inhalt des Prozessorstatusregisters auf den Stack entsprechend der Position des Stackpointers. Das Statusregister selbst bleibt unverändert. Der Stackpointer wird um eins vermindert. PLP überträgt umgekehrt den gemäß Stackpointer nächsten Wert des Stacks in das Prozessorstatusregister. Alle Flags des Statusregisters werden dabei entsprechend der Bitkombination des eben übertragenen Wertes beeinflußt. Der Stackpointer wird um eins erhöht.
PHP und PLP sind 1-Byte-Befehle mit impliziter Adressierung.

Insgesamt existieren zehn Assemblerbefehle zur Beeinflussung des Stacks bzw. Stackpointers:

JSR, RTS, PHA, PLA, PHP, PLP, TXS, TSX, BRK und RTI.

BRK und RTI hängen bereits direkt mit unserem nächsten Thema zusammen, der „Interrupt-Verarbeitung". Doch bevor wir dazu kommen, ist es jetzt an der Zeit, uns einmal ein Gesamtbild des Aufbaus des Mikroprozessors anzusehen. Mit dem Stackpointer haben wir das letzte dazu notwendige uns bisher noch fehlende Register kennengelernt.



Zeichenerläuterungen:

X = X-Indexregister, (1 Byte)
Y = Y-Indexregister, (1 Byte)
P = Prozessorstatusregister, (1 Byte)
S = Stackpointer (Stapelzeiger), (1 Byte)
ALU = Arithmetisch/logische Einheit („arithmetic/ logical unit"), (1 Byte)
A = Akkumulator, (1 Byte)
PC = „program counter", Programmzeiger, (2 Byte)

da = Datenleitung
ad = Adressleitung

X, Y, P, S und A sind wohl nicht weiter erklärungsbedürftig. ALU ist die interne Recheneinheit des Prozessors. Sie wird sowohl zur Berechnung von Daten – zum Beispiel bei ADC, AND, LSR etc. – als auch von Adressen – zum Beispiel bei BCC, BNE, BEQ etc. – benutzt. Die Berechnungen erfolgen in direktem Austausch mit dem Akkumulator, bei Adressen auch mit dem PC. Die ALU wird intern verwaltet, der Programmierer kann nicht direkt auf sie zugreifen.
Der PC schließlich ist ein 2-Byte-„Zeiger", bestehend aus zwei Bytes, die als LSB und MSB auf die aktuelle Adresse der Programmausführung zeigen. Wenn zum Beispiel der Befehl „JMP 8000" abgearbeitet wird, so wird dadurch der PC so gestellt, daß er auf Adresse 8000 zeigt (s. auch Seite 5-5). Akkumulator und ALU sowie der PC stehen direkt miteinander in Verbindung.

Ein Wort noch zu dem Begriff „Datenbus”. Unter einem „Bus” versteht man – in diesem Zusammenhang! – eine Anordnung von elektrischen Leitungen, im Unterschied zu einer einzelnen Leitung. Der Datenbus unseres Prozessors ist „acht Bit breit”, d.h., er besteht aus acht Leitungen. Der Adressbus ist „sechzehn Bit breit”. Die Bezeichnung „Bus” ist also durchaus der des täglichen Lebens entlehnt, schließlich können in einem Autobus auch mehrere Passagiere mitfahren.
Wenn also ein Byte zum Beispiel von Akkumulator in das X-Register übertragen werden soll, dann werden alle acht Bits über acht Leitungen parallel übermittelt. Mehrere Bytes hingegen können – auf dem Datenbus – nicht gleichzeitig übertragen werden, sondern nur in Serie nacheinander. Man spricht hier auch von „bit-paralleler” und „byte-serieller” Übertragung.

So, diesen Überblick über den Gesamtaufbau des Prozessors glaubten wir Ihnen schuldig zu sein, wo wir doch zuvor schon so viel über die einzelnen Komponenten gesprochen hatten. Doch jetzt wollen wir uns dem Thema „Interrupt” zuwenden.

„Interrupt” heißt übersetzt „Unterbrechung” und meint eine Unterbrechung der normalen Arbeitsweise des Prozessors. Bisher sind wir immer von folgendem ausgegangen: Der Mikroprozessor bearbeitet unser Programm, so wie wir es vorgeben, Befehl für Befehl. Änderungen in der Ablauffolge ergeben sich durch Sprungbefehle innerhalb des Programms. Auf externe Ereignisse – wie zum Beispiel das Drücken einer Taste – kann unser Programm nur reagieren, wenn wir eine entsprechende Abfrage eingebaut haben und diese Abfrage auch zum richtigen Zeitpunkt abgearbeitet wird – nämlich genau dann, wenn die Taste gedrückt wird. Die sich daraus ergebende Problematik in der Praxis ist klar: Entweder fragen wir laufend die Taste ab, wodurch die Arbeitsgeschwindigkeit des übrigen Programms erheblich verlangsamt wird oder aber wir lassen es darauf ankommen, den einen oder anderen Tastendruck vom Programm aus zu verpassen. Selbst wenn wir den Tastendruck durch eine externe elektronische Schaltung speichern, kann ihn unser Programm immer nur mit Verzögerung verarbeiten.

Um diesem Problem zu begegnen, hat unser Mikroprozessor zwei sog. „Interrupt-Leitungen”, IRQ und NMI. „IRQ” steht für „interrupt request”, übersetzt „Interrupt-Anfrage”, „NMI” bedeutet „non-maskable interrupt”, „nicht-maskierbarer Interrupt”. Sie werden die Bedeutungen in Kürze besser verstehen.

Sobald an einer Interrupt-Leitung ein Aktiv-Signal von einem externen Gerät – zum Beispiel unserer Taste – gemeldet wird, bricht der Prozessor die normale Programmausführung ab und springt an eine Stelle, die vorher festgelegt wird. Ab dieser Adresse steht dann im allgemeinen ein Programmteil, der den Interrupt „bedient”, der also zum Beispiel auf den Tastendruck reagiert. Die Festlegung der Adresse, ab welcher der Interrupt bedient wird, geschieht in zwei aufeinanderfolgenden Bytes als LSB und MSB. Wir nennen diese zwei Bytes zusammen „Interrupt-Vektor”.

Der IRQ-Vektor liegt in den Adressen 65534 (LSB) und 65535 (MSB), der NMI-Vektor in den Adressen 65530/65531. Tritt nun ein Interruptimpuls an der IRQ-Leitung des Prozessors auf, so geschieht im einzelnen folgendes:

Der Prozessor fragt das „Interrupt-Disable-Flag“ (vermeide Interrupt“) des Statusregisters auf seinen Zustand ab. Ist dieses Flag gesetzt, so wird der Interrupt nicht berücksichtigt, die normale Programmabarbeitung läuft weiter.

Andernfalls wird der Programmablauf unterbrochen und in dieser Reihenfolge das LSB des PC, das MSB des PC und der Inhalt des Prozessorstatusregisters auf den Stack gelegt und der Stackpointer entsprechend um drei vermindert.
Die „Rettung“ von PC (Programmzähler) und Statusregister ist wichtig, damit zu einem späteren Zeitpunkt, wenn der Interrupt fertig bedient worden ist, die Programmausführung an der unterbrochenen Stelle wieder aufgenommen werden kann.

Wenn PC und Statusregister auf dem Stack abgelegt sind, prüft der Prozessor, welche Adresse in den Speicherstellen 65534 und 65535 als LSB/MSB gespeichert ist und gibt die Programmabarbeitung an die dort angegebene Adresse weiter.

Es ist nun Sache des Programmierers, dafür zu sorgen, daß sein Programm auf den Interrupt richtig reagiert, daß zum Beispiel beim Drücken der Taste „A“ der Buchstabe „A“ auf dem Bildschirm dargestellt wird. Man nennt diesen Programmteil „Interrupt-Routine“.

Bei einem Interruptsignal an der NMI-Leitung passiert im Grunde das gleiche wie bei IRQ, mit zwei Abweichungen; erstens entfällt bei NMI die Prüfung des „Intgerrupt-Disable-Flag“ im Statusregister, der Interrupt wird in jedem Fall ausgeführt und zweitens liegt der NMI-Vektor in den Speicherstellen 65530/65531.

Gleichgültig, ob es sich um einen IRQ- oder einen NMI-Interrrupt handelt, muß die Interrupt-Routine in jedem Fall mit dem Befehl RTI („return from interrupt“, „Rückkehr aus Interruptroutine“) beendet werden. RTI nimmt den bei Interruptbeginn auf dem Stack abelegten Programmzählerinhalt sowie das Statusbyte wieder vom Stack herunter und überträgt sie in den PC bzw. das Prozessorstatusregister. Damit wird wieder der Zustand vor Ausführung der Interruptroutine hergestellt und die normale Programmabarbeitung wird an der Stelle wieder aufgenommen, an der sie vorher abgebrochen wurde.

Das ähnelt ein bißchen der Unterprogramm-Programmierung. Nur wird bei der Interrupt-Verarbeitung eben statt eines Unterprogramms eine Interrupt-Routine angesprungen. Der Aufruf dieser Routine erfolgt auch nicht vom Programm aus, sondern von außen durch ein elektrisches Signal. Die Aufrufadresse ist im Interrupt-Vektor festgelegt. Beendet wird die Interrupt-Routine mit RTI, was dem RTS-Befehl bei Unterprogrammtechnik entspricht.

Während JSR aber nur zwei Bytes nämlich die Rücksprungadresse als LSB und MSB auf dem Stack ablegt, sind es bei einem Intgerrupt drei Bytes Programmzähler als LSB und MSB (entspricht praktisch der Rücksprungadresse) und Statusbyte.

RTI ist ein 1-Byte-Befehl mit impliziter Adressierung. Alle Flags des Statusregisters werden entsprechend dem vom Stack geholten Statusbyte beeinflußt.

Bei einem Interrupt werden PC und Statusregister automatisch „gerettet“, da diese Angaben zur Fortführung des normalen Programms nach Bedienung des Interrupts unbedingt erforderlich sind. Wenn Sie aber in Ihrem Programm noch weitere Register „retten“ wollen, dann muß dies in der Interrupt-Routine programmiert werden. Der Anfang einer solchen Routine, bei der Akkumulator sowie X- und Y-Register auf den Stack geschoben werden, kann zum Beispiel wie folgt aussehen:

```
PHA     ; legt Akku auf Stack ab.
TXA     ; Legt Inhalt des X-Registers
PHA     ; auf den Stack.
TYA     ; Legt Inhalt des Y-Registers
PHA     ; auf den Stack.
```

Die hier gezeigte Methode, die zu rettenden Register auf dem Stack zwischenzuspeichern, ist üblich. Das Ende der Interrupt-Routine wird so oder ähnlich aussehen:

```
PLA     ; Holt Inhalt des Y-Registers
TAY     ; vom Stack.
PLA     ; Holt Inhalt des X-Registers
TAX     ; vom Stack.
PLA     ; Holt Inhalt des Akku vom Stack.
```

Wir haben bereits bei der Besprechung des Ablaufs eines IRQ-Interrupts des Bedeutung des I-Flags im Statusregister („interrupt disable“) erklärt. Bei gesetztem Flag wird ein Aktivsignal an der IRQ-Leitung des Prozessors nicht berücksichtigt. Das mag zum Beispiel wichtig sein für einen Programmteil, der möglichst schnell ablaufen soll. Hier würde ein Interrupt nur unnötige Verzögerungen verursachen. Also wird man zu Anfang des zeitkritischen Programmteils das I-Flag setzen und danach wieder löschen. Dazu existieren zwei Anweisungen, SEI und CLI.

SEI („set interrupt disable, „setze das Interrupt-unwirksam-Flag“) setzt das I-Flag und CLI („clear interrupt disable“, „lösche Interrupt-unwirksam-Flag“) löscht es. Das I-Flag wirkt ausschließlich auf den IRQ-Interrupt, hingegen nicht auf den NMI-Interrupt. Schließlich steht ja „NMI“ für „nicht maskierbarer Interrupt“. Liegen NMI und IRQ gleichzeitig an, hat NMI in jedem Fall die höhere Priorität.

Wenn durch ein Aktivsignal auf der IRQ-Leitung des Prozessors ein Interrupt ausgelöst wird, so setzt das gleichzeitig das I-Flag im Statusregister. Während der Interrupt bearbeitet wird, können also keine weiteren Interrupt-Anforderungen angenommen werden. Sollen dennoch schon während der Interrupt Bedienung weitere Interrupts möglich sein, so müssen Sie als Programmierer mit CLI das I-Flag löschen. Auf diese Weise können verschachtelte Interrupts entstehen, ähnlich wie das auch bei Unterprogrammen der Fall sein kann.

Im allgemeinen sind externe Geräte nicht direkt mit dem Mikroprozessor verbunden, sondern über „Interface-Bausteine". Als „Interface" bezeichnet man die Schnittstelle zwischen zwei getrennten Einheiten zum Beispiel einer Taste und dem Mikroprozessor. Das Interface dient der „Kommunikation" der Einheiten. (Nebenbei bemerkt, so erklärt sich auch unser Firmenname „INTERFACE AGE", frei übersetzt „Kommunikationszeitalter". Wir hoffen, damit eine uns häufig gestellte Frage beantwortet zu haben.)
Es würde zu weit führen, hier alle oder auch nur einige der verfügbaren Interface-Bausteine zu besprechen. „Interfacing" ist ein derart komplexes Thema, daß dazu ein eigenes Buch mindestens vom Umfang des vorliegenden notwendig wäre.

Wir wollen aber jetzt noch eine Möglichkeit der Interrupt-Programmierung besprechen, die nicht auf Signale von externen Geräten angewiesen ist. Mit dem Assemblerbefehl BRK („break", „Unterbrechung") läßt sich nämlich ein Interrupt von einem Programm aus aktivieren.

BRK löst einen Interrupt aus, der einer Aktiv-Meldung der IRQ Leitung des Prozessors entspricht. BRK ist ein 1-Byte-Befehl mit impliziter Adressierung, der Assemblercode für BRK ist 0. Läuft der Prozessor nun im Programmverlauf auf einen BRK-Befehl auf, so geschieht im einzelnen folgendes (Vergleichen Sie mit dem Ablauf bei einem Interruptimpuls an der IRQ-Leitung, S. 9-17):

(Das I-Flag des Statusregisters bleibt unberücksichtigt. Der Interrupt wird in jedem Fall ausgeführt.)

Der Programmzählerstand des zweiten Bytes nach dem BRK-Befehl wird als LSB und MSB auf den Stack gelegt.

Das B-Flag des Prozessorstatusregisters wird auf 1 gesetzt. Dann wird das Statusbyte ebenfalls auf den Stack gebracht.

Dann wird der Programmablauf an die im IRQ-Vektor (Adressen 65534/65535) festgelegte Adresse abgegeben.

Dadurch, daß das B-Flag bei einem durch die BRK-Anweisung ausgelösten Interrupt gesetzt wird, kann dieser Vorfall in der Interrupt-Routine leicht von einem von außen kommenden Interrupt unterschieden werden. Beachten Sie, daß die Rücksprungadresse auf das zweite Byte nach dem BRK-Befehl zeigt. Soll das Programm direkt mit dem Byte nach BRK weitergeführt werden, so müssen Sie dies in der Interrupt-Routine durch entsprechende Stack-Manipulationen programmieren.

Die wohl häufigste Verwendung für BRK ist das Austesten eines Programmes. Wenn Sie feststellen, daß in einem Assemblerprogramm noch ein Fehler ist, ohne diesen genauer loklisieren zu können, können Sie BRK an die Stelle eines bereits existierenden Befehls einsetzen. Das Programm hält dann beim Testlauf an dieser Stelle an und springt in die Interrupt-Routine, wo Sie sich Register ausgeben lassen können oder die Inhalte von Speicherzellen überprüfen können usw. ‚T.EX.AS.' zum Beispiel benutzt ein ähnliches Prinzip zum Ablauf des Simulators im Direktassemblerteil.

Sie haben gelernt, daß der IRQ-Vektor also die Speicherstellen, in denen steht, wo die Interrupt-Routine beginnt in den Adressen 65534 und 65535 liegt. Nun ist es ja aber bei den meisten Mikrocomputern so, daß diese Adressen sich im ROM Bereich befinden. Wie Sie wissen, kann eine ROM-Speicherzelle nicht durch einfaches Überschreiben mit einem anderen Wert geändert werden.

Die meisten Mikrocomputer sind aber derart ausgelegt, daß ca. alle 1/60 Sekunde ein Interrupt durch den Computer selbst ausgelöst wird. In der Interrupt-Routine wird die Tastatur abgefragt, ob eine Taste gedrückt wird, der Bildschirm bedient, eine evtl. vorhandene interne Uhr weitergezählt und vieles andere. Der Anfang dieser Standard-Interrupt-Routine steht bei praktisch allen Mikrocomputern als Zwei-Byte-Vektor im RAM. Sie erinnern sich RAM ist der Speicherbereich, der gelesen und in den geschriebenen werden kann. In welchen Speicherstellen genau der Vektor liegt, hängt vom Betriebssystem ab und ist von Computer und Computer verschieden. Indem Sie als Programmierer nun diesen Vektor im RAM auf Ihre eigene Interrupt-Routine zeigen lassen, können Sie veranlassen, daß alle 1/60 Sekunde Ihr Programmteil ausgeführt wird. Danach kann zur Standard-Interrupt-Routine weiterverzweigt werden.

Das zeigt, wie wichtig es für fortgeschrittene Programmierung ist, eben nicht nur den Befehlsvorrat des Prozessors zu beherrschen, sondern auch das Betriebssystem des jeweiligen Computers zu kennen und zu verstehen. Wir werden im nächsten Kapitel etwas näher darauf eingehen.

So, damit sind wir am Ende dieses Kapitels angelangt. Sie kennen jetzt ausnahmslos alle Anweisungen des 6502-Mikroprozessors sowie die damit verbundenen Programmiermöglichkeiten. Dabei haben wir insbesondere in diesem Kapitel einige Methoden besprochen, die bereits einem mittleren Fortgeschrittenheitsgrad entsprechen. Zu einem erfolgreichen Programmieren gehört aber immer auch die Praxis. Wir möchten Sie an dieser Stelle noch einmal ausdrücklich ermuntern, selbst zu experimentieren. Mit ‚T.EX.AS.' steht Ihnen dazu ein extrem leistungsfähiges und professionelles Assembler-Entwicklungs-System zur Verfügung. Für den Anfang ist es sicherlich auch sinnvoll, wenn Sie sich andere in Assembler geschriebene Programme ansehen und daraus lernen. Solche Programme möglichst kommentiert finden Sie in Fachzeitschriften, in Applikations-Büchern und anderen Publikationen.

Zusammenfassung

An „Boole'schen Operationen" existieren die Assemblerbefehle AND, ORA und EOR mit den verschiedensten Adressierungsarten. Der Akkumulator wird entsprechend der Boole'schen Definition mit dem Argument verknüpft, das Ergebnis steht im Akkumulator. Die Definitionen sind im Kapitel erklärt.
BIl wirkt wie AND, speichert das Ergebnis aber nicht im Akkumulator ab. Das Statusregister wird entsprechend beeinflußt.

Die folgenden Befehle dienen zum Verschieben bzw. Rotieren der einzelnen Bits innerhalb eines Registers oder einer Speicherselle. Das Carry-Bit wird dabei unabhängig von der Adressierungsart wie ein neuntes Bit behandelt.

ISR	Rechts-Verschiebung
ASI	Links-Verschiebung
ROR	Rechts-Rotation
ROL	Links-Rotation

Im Adressbereich von 256 bis 511 (Page 1) liegt der Stack. Er funktioniert nach dem Stapelspeicherprinzip „last in – first out". Der Stack wird von oben gefüllt, die aktuelle Position ist durch den Stackpointer (Stapelzeiger) bestimmt. Das Einrichten des Stackpointers kann mit TXS und TSX erfolgen.

Der Stackpointer wird von folgenden Befehlen erniedrigt: JSR, PHA, PHP, BRK und von diesen erhöht: RTS, PLA, PLP, RTI.

JRS	Unterprogrammsprung
RTS	Unterprogramm beenden
PHA	Akkumulatorinhalt auf Stack legen
PLA	Akkumulatorinhalt von Stack holen
PHP	Prozessorstatusregister auf Stack legen
PLP	Prozessorstatusregister von Stack holen
BRK	Interrupt auslösen (IRQ)
RTI	Interrupt beenden (IRQ und NMI)

Der Prozessor hat zwei Interrupt-Leitungen (IRQ und NMI), die bei aktivem externem Signal den normalen Ablauf eines Programms unterbrechen und zu durch sog. Interrupt-Vektoren festgelegten Interrupt-Routinen verzweigen. Bei Interrupt-Auslösung werden der Programmzähler als LSB und MSB sowie das Statusbyte auf den Stack „gerettet". RTI beendet eine Interrupt-Routine und holt den Programmzählerinhalt und das Statusbyte wieder vom Stack, um die Programmausführung an der unterbrochenen Stelle fortzusetzen. IRQ-Vektor: 65534/65535 (LSB/MSB), NMI-Vektor: 65530/65531 (LSB/MSB).

Der IRQ-Interrupt wird über das I-Flag gesteuert (SEI, CLI). Bei gesetztem I-Flag (SEI) werden alle IRQ-Interrupts unterdrückt, andernfalls (CLI) durchgeführt. Der NMI ist nicht unterdrückbar.

BRK bewirkt eine Interrupt-Auslösung vom Programm her und wirkt wie ein entsprechendes Signal an der IRQ-Leitung.

Kapitel 10: Betriebssystemroutinen

Wie Sie wissen, ist es zur fortgeschrittenen Programierung wichtig, das Betriebssystem des jeweiligen Computers zu kennen. Das Betriebssystem ist ein Assemblerprogramm, das bei Einschalten des Computers sofort in Aktion tritt und für die Kommunikation (Tastatur, Bildschirm, einfache Befehle) mit dem Benutzer sorgt.

Die Programmiersprache – zum Beispiel BASIC – gehört nicht mehr zum Betriebssystem – korrekt genommen. Bei vielen Mikrocomputern sind Betriebssystem und Sprache jedoch so stark vermischt, daß eine Trennung praktisch unmöglich ist. Bei anderen hingegen wird dies ganz deutlich unterschieden.

Wenn Sie professionell in Assembler programmieren wollen, sollten Sie sich ein kommentiertes! Listing des Betriebssystems Ihres Computers zulegen. Prüfen Sie vor dem Kauf die Ausführlichkeit und Güte der Kommentierung, die Unterschiede sind oft beträchtlich.

Wir wollen nun in diesem Kapitel einige wenige aber prägnante Möglichkeiten des Betriebssystems und des BASIC-Interpreters erläutern, die bei den meisten Mikrocomputern mit 6502 oder ähnlichem Mikroprozessor anzutreffen sind. Dabei geht es nur darum, Ihnen einmal exemplarisch darzustellen, wie Sie damit progrmmieren können, eine auch nur annähern vollständige Auflistung der Betriebssystemroutinen würde den Rahmen dieses Buches sprengen.

Wenn Sie genauer darüber nachdenken, wird Ihnen auch sehr schnell klar werden, für welche Aufgaben das Betriebssystem Routinen beinhalten muß. Da sind zum Beispiel „Ausgabe eines Zeichens", „Annahme eines Tastendrucks", „Auswertung eines numerischen Ausdrucks", „Auswertung eines Stringausdrucks", „Test auf Klammer auf", „Test auf Klammer zu", „Test auf Komma", „Test auf Doppelpunkt", „Prüfung auf numerische Eingabe", „Prüfung auf Stringeingabe" und viele viele andere. Diese Routinen sind im allgemeinen so konzipiert, daß Sie sie von Ihrem Programm aus mittels JSR als Unterprogramm aufrufen können. Natürlich muß man dazu wissen, an welchen Adressen diese Routinen liegen und welche Parameter wie übergeben werden. Das eben steht in einem guten Betriebssystm-Listing.

Wir wollen uns einmal eine Routine heraussuchen, die bei allen gängigen Mikrocomputern mit 6502-Prozessor und Microsoft-BASIC zu finden ist (Commodore, Apple, Atari etc.). Genannt wird die Routine zumeist „CHRGET", was daher kommt, daß viele Assembler-Systeme nur Labels mit bis zu sechs Stellen zulassen. „CHRGET" steht für „character get", wörtlich übersetzt „hole Zeichen". Die CHRGET-Routine dient dazu, ein einzelnes Zeichen aus dem Programmtext oder dem sog. „BASIC-Eingabepuffer" zu holen und auf ganz bestimmte Kriterien hin zu überprüfen. Wir werden sehen, daß sich diese Kriterien relativ leicht ändern lassen, so daß wir die CHRGET Routine für unsere Zwecke „umbauen" können.

Die Situationen, in der die CHRGET-Routine von veschiedenen Programmteilen aus aufgerufen wird, sind vielfältig.
Einmal im Direktmodus, d.h., wenn kein Programm abläuft, sondern Sie direkt etwas eintippen. In dem Moment, in dem Sie die Direkteingabe beenden durch Drücken der RETURN- oder ENTER-Taste wird der eingegebene Text in den sog. „BASIC-Eingabepuffer“ gelegt. Damit nun das Betriebssystem bzw. der BASIC-Interpreter etwas damit anfangen kann, muß der BASIC-Eingabepuffer Zeichen für Zeichen überprüft und verarbeitet werden. Dazu wird der Textzeiger der CHRGET-Routine auf den Anfang des BASCIC-Eingabepuffes gestellt und danach wird ein Zeichen nach dem anderen durch Weiterzählen des Zeigers geholt. Die Weiterverarbeitung geschieht nicht mehr in der CHRGET-Routine, sondern in anderen Teilen des Betriebssystems.

Die CHRGET-Routine wird aber auch benötigt, wenn ein BASIC-Programm abläuft. In diesem Fall wird der Textzeiger der CHRGET-Routine nicht auf den BASIC-Eingabepuffer gesetzt, sondern auf den Anfang des Programmtextes. Durch Weiterzählen des Zeigers unter Kontrolle anderer Betriebssystemteile wird das BASIC-Programm Zeichen für Zeichen gelesen und zur Verarbeitung weitergegeben. Dabei sind natürlich auch Sprunganweisungen usw. zu beachten.

Bei den Mikrocomputern, die den BASIC-Interpreter im ROM haben und nicht erst von Diskette laden müssen, befindet sich natürlich auch die CHRGET-Routine im ROM. Sie wird aber beim Einschalten des Computers in den RAM-Bereich kopiert, je nach Comptertyp an unterschiedliche Stellen, jedoch immer in die Zero-Page; Sie erinnern sich, es gibt spezielle Zeropage-Adressierungsarten, die platzsparender und insbesondere im Ablauf schneller sind. Schließlich wird die CHRGET-Routine ständig benötigt.
Der Grund aber, warum die CHRGET-Routine in den RAM-Bereich kopiert wird, ist, daß zwei Bytes dieser Routine laufend geändert werden, was ja im ROM nicht möglich ist. Wir werden gleich sehen, wie das vor sich geht.

Hier nun die CHRGET-Routine am Beispiel des Commodore CBM 8032 für andere Computertypen sieht die Routine praktisch genauso aus, liegt aber u.U. an anderen Adressen.

Adresse	Assembler-Code	Quelltext
112	230 119	INC 119
114	208 2	BNE 118
116	230 120	INC 120
118	173	LDA Textzeiger
121	201 58	CMP #58
123	176 10	BCS 135
125	201 32	CMP #32
127	240 239	BEQ 112
129	56	SEC
130	233 48	SBC #48
132	56	SEC
133	233 208	SBC #208
135	96	RTS

Bei der hier gezeigten Darstellung wird zusätzlich zu dem Assembler-Programm („Quelltext") auch noch der Assembler-Code jedes einzelnen Befehls bzw. jeder Adresse angegeben. Das ist durchaus üblich. Wenn Sie aber das Programm mit ‚T.EX.AS.' eingeben wollen, brauchen Sie natürlich nur den Quelltext einzutippen, die Assembler-Codes errechnet sich ‚T.EX.AS.' selbständig, sie werden gar nicht erst angezeigt.

Die zwei Bytes 119 und 120 sind mit „..." gekennzeichnet, da sie laufend geändert werden. Sie stellen nämlich den bereits angesprochenen Textzeiger dar. Aber gehen wir in der Erläuterung der CHRGET-Routine der Reihe nach vor.

Die ersten drei Anweisungen – ‚INC/BNE/INC' – erhöhen die Speicherstellen 119 und 120 um genau eins. Dabei wird Adresse 119 als LSB und Adresse 120 als MSB behandelt. Das LSB wird immer wieder von 0 bis 255 gezahlt (Sie erinnern sich, statt 256 beginnt INC automatisch wieder bei 0). Wenn dabei ein Wert von 1 bis 255 anfällt, so wird die zweite INC-Anweisung durch ‚BNE 118' übersprungen,das MSB bleibt in diesem Fall also unverändert. Aber jedesmal, wenn das LSB bei 0 angelangt ist, bleibt die Bedingung BNE erfüllt und es wird ‚INC 120' ausgeführt, das MSB wird um eins erhöht. Hier wird also eine Zwei-Byte-Adresse (119/120) korrekt als LSB und MSB inkrementiert, ein „Trick", den Sie sich merken sollten.

Nun wird der Inhalt der um jeweils eins erhöhten Adresse („TEXTZEIGER") mit LDA in den Akkumulator geladen. Der Textzeiger „wandert" folglich Zeichen für Zeichen über den zu lesenden Text BASIC-Eingabepuffer oder BASIC-Programmtext und lädt das jeweils aktuelle Zeichen in den Akkumulator.

Hierbei wird die Tatsache ausgenutzt, daß in Assembler kein grundsätzlicher Unterschied zwischen Programm und Daten besteht – wir haben bereits in Kapitel 5 ausführlich darüber gesprochen. Einmal ist der TEXTZEIGER Teil des Programms, zum anderen wird er als zwei Datenbytes laufend verändert. Diese Programmier-Methode ist unschön und wird deshalb auch von vielen Programmierern abgelehnt, sie ist aber recht praktisch wie die CHRGET-Routine beweist. Wir empfehlen Ihnen allerdings, in Ihrem eigenen Programm strenger zwischen Programm und Daten zu unterscheiden und im allgemeinen keine Programme zu schreiben, die sich selbst verändern. Über ein solches Program die Übersicht zu behalten, ist nämlich äußerst schwierig, wie wollen Sie Programmteile dokumentieren, die sich selbständig verändern?
Immerhin ist die Ausnahme bei der CHRGET-Routine sinnvoll. Andernfalls müßte der Textzeiger mittels einer Indizierung etwa ‚LDA (TEXTZEIGER),Y' programmiert – werden, was nicht nur umständlich ist, sondern auch bei der Ausführung viel Zeit verbraucht. Beachten Sie, daß bei einer Indizierung der Label „TEXTZEIGER" innerhalb Klammern steht („indirekt"), der eigentliche Textzeiger ist also nicht Teil des Programmes, sondern steht in einem Datenspeicher. Es wird nicht das Programm selbst verändert, sondern eine Speicherzelle außerhalb.

Damit haben wir den ersten Teil der CHRGET-Routine, der die eigentliche Funktion des Zeichen-Lesens beinhaltet, bereits besprochen. Die auf LDA folgenden Anweisungen dienen der Abfrage auf bestimmte Kriterien.

Der erste Vergleich erfolgt mit dem Wert 58 (,CMP #58'). Im Falle dadurch das Carry-Flag gesetzt wird (BCS), erfolgt ein Sprung zu Adresse 135 (RTS). Das bedeutet, daß die CHRGET-Routine verlassen wird, wenn der Inhalt des Akkumulators gleich oder größer als 58 ist. Dann sind keine weiteren Prüfungen mehr nötig. 57 ist genau die obere Grenze des Zahlen-Bereichs (Ziffern 0 bis 9) des ASCII-Codes. Zeichen, deren ASCII-Wert größer als 57 ist (das ist dasselbe wie „gleich oder größer als 58") sind nicht-numerische Zeichen wie Buchstaben, „Token" (verschlüsselte BASIC-Worte), Sonderzeichen usw. Solche Zeichen werden mithin im Akkumulator ohne weitere Test an die Routine weitergegeben, die die CHRGET-Routine aufgerufen hat.

Ist der ASCII-Code des Zeichens kleiner als 58, so wird die CHRGET-Routine mit ,CMP #32' fortgesetzt. 32 ist der ASCII-Code für ein Leerzeichen („Space"). Wenn es sich um ein solches handelt, so bleibt es unberücksichtigt und es wird zum Anfang der CHRGET-Routine gesprungen. Leerzeichen werden also in der CHRGET-Routine abgeblockt.

Die restlichen Prüfungen der CHRGET-Routine dienen dazu, zwischen den Ziffern (ASCII-Codes 48 bis 57) und den restlichen Zeichen mit Codes von 0 bis 47 (ausgenommen 32, s.o.) zu unterscheiden. Die beiden dazu notwendigen Subtraktionen mit SEC und SBC sind äußerst trickreich programmiert. Bei Ziffern wird das Carry-Flag gelöscht, andernfalls wird es gesetzt. Wie funktioniert das?

Nun, 48 ist der ASCII-Code für die Ziffer 0, 49 der für die 1 usw. bis 57 für 9. Dieser Bereich soll selektiert werden. Nun wird in Adresse 129 mit SEC das Carry-Flag gesetzt. Danach wird 48 vom Akkumulatorinhalt subtrahiert. Ist der Inhalt des Akkumulators 48 oder größer, so bleibt das Carry-Flag unverändert gesetzt – das Zeichen ist numerisch. Andernfalls ist das Ergebnis „negativ" und das Carry-Flag wird gelöscht (s. Kapitel 6). Bedenken Sie, daß der Akkumulator zu diesem Zeitpunkt den um 48 verminderten Wert enthält.

Es folgt die zweite Subtraktion. In Adresse 132 wird wiederum das Carry-Flag gesetzt, wie das vor Subtraktionen erforderlich ist. Und nun wird der Wert 208 vom Inhalt des Akkumulators abgezogen. Und damit sind wir auch schon bei der äußerst trickreichen Programmierung dieses Teils. Erst wird 48, dann 208 subtrahiert. 48 plus 208 ergeben aber genau 256, es wird mithin insgesamt 256 subtrahiert. Sie wissen aber, daß 256 in einem 8-Bit-Register, wie es eben der Akkumulator darstellt, gleich 0 ist. Alles in allem wird also 0 subtrahiert, oder anders ausgedrückt, der Inhalt des Akkumulators bleibt unverändert. Die Subtraktion von 208 gleicht also die vorangegangene von 48 wieder aus, bewirkt aber, daß bei ASCII Codes von 0 bis 48 das Carry-Flag gelöscht wird.

Die Subtraktion von 208 gleicht also die vorangegangene von 48 wieder aus. Der Sinn dieses Abschnitts besteht ausschließlich darin, das Carry-Flag entsprechend zu setzen oder zu löschen.

Nach der ersten Subtraktion enthält der Akkumulator entweder einen Wert von 0 bis 9 bei gesetztem Carry (entspricht den Ziffern 0 bis 9) oder aber einen Wert von 208 bis 255 bei gelöschtem Carry (Alphabet, Sonderzeichen; Sie wissen, daß Werte kleiner als 0 wieder von 255 an gerechnet werden.).
Im ersten Fall (Akkumulatorinhalt 0 bis 9) wird durch die zweite Subtraktion von 208 das Carry-Flag gelöscht, da das Ergebnis negativ wird. Im zweiten Fall (Akkumulatorinhalt 208 bis 255) hingegen bleibt das Carry-Flag gesetzt, da die Subtraktion von 208 im positiven Bereich verbleibt.
Und so ergibt sich, daß bei numerischen Zeichen die CHRGET-Routine das Carry-Flag löscht, während sie es bei Buchstaben, Sonderzeichen etc. setzt.

Dieses Beispiel zeigt aber auch sehr anschaulich, mit welchen „Tricks" oftmals in der Assembler-Programmierung gearbeitet wird. Genau genommen werden hier natürlich nur Konsequenzen genutzt, die sich aus dem bisher Besprochenen klar ergeben.

Damit sind wir auch schon am Ende der CHRGET-Routine angelangt. Zusammenfassend können wir sagen:

CHRGET übergibt Zeichen mit einem ASCII-Code größer als 58 direkt weiter, das Carry-Flag ist in diesem Fall gesetzt. Leerzeichen (ASCII-Code 32) werden unterdrückt. Bei Zeichen mit einem ASCII-Code im Bereich von 48 bis 57 (Ziffern) wird das Carry-Flag gelöscht, bei Zeichen mit einem ASCII-Code im Bereich von 0 bis 47 wird das Carry-Flag gesetzt. Die CHRGET-Routine wird als ein Unterprogramm zum Einlesen eines einzelnen Zeichens aus einem BASIC-Text benutzt.

Nachdem wir nun die CHRGET-Routine so genau kennengelernt haben, wollen wir jetzt eine Möglichkeit besprechen, wie wir unser Wissen gut für eigene Programme ausnutzen können. Es gibt für viele Mikrocomputer eine Vielzahl von Programmierhilfen, die das BASIC um Befehle erweitern. Diese „Tool-Kits" sind im allgemeinen nach einem recht einfachen Prinzip aufgebaut, das die CHRGET-Routine „anzapft". Das heißt, es werden in die CHRGET-Routine Abfragen eingebaut, die bei bestimmten Zeichen oder Zeichenkombinationen (z.B. Befehlsworte) zu eigenen Routinen des betreffenden „Tool-Kits" verzweigen und eine Sonderfunktion auslösen. Bei zahlreichen Kits wird ausschließlich diese sehr einfache Möglichkeit verwendet. Sie erkennen das daran, daß die erweiterten Befehle lediglich im Direktmodus, jedoch nicht innerhalb eines Programms, zur Verfügung stehen. Wir wollen jetzt diese einfache Möglichkeit kennenlernen.

Da wir die CHRGET-Routine selbst nicht beliebig verlängern können, fügen wir einfach einen Sprungbefehl ein, der an den Anfang unserer eigenen Abfrage-Routine verzweigt.

Die „Anzapfung“ der CHRGET-Routine kann zum Beispiel wie folgt aussehen:

```
Adresse Assembler-Code          Quelltext

112     230  119                INC    119
114     208    2                BNE    118
116     230  120                INC    120
118     173  . . . . . .        LDA  TEXTZEIGER
121            24   121         JMP  31000
124     234                     NOP
125     201   32                CMP   # 32
127     240  239                BEQ    112
 .       .                       .
 .       .                       .
```

Der Rest der CHRGET-Routine bleibt unverändert. Dabei haben wir angenommen, daß unsere Routine bei Adresse 31000 beginnt. Das Ende des BASIC-Bereiches muß also bei 31000 begrenzt werden. Bei dem hier modellhaft besprochenen Commodore CBM 8032 wird die BASIC-Obergrenze als LSB und MSB in den Adressen 52 und 53 festgelegt. Wir werden gleich darauf zurückkommen.

Zuvor wollen wir jedoch unsee „BASIC-Erweiterung“ bei Adresse 31000 weiterentwickeln. Zunächst wird auf das neue Befehlswort, das der Computer von jetzt an vestehen soll, verglichen. Der Einfachheit halber nehmen wir als „Befehlswort“ das Zeichen „!“. Da unser „Anzapfung“ mit „JMP 31000“ den CMP-Vergleich mit 58 in der CHRGET-Routine überschrieben hat, müssen wir ihn in unserer Routine berücksichtigen.

```
31000    CMP    # 33     ; Vergleich mit „!“
31002    BEQ  31011      ; Ja, dann zu „!“-Routine
31004    CMP    # 58     ; ersetzt Abfrage der CHRGET-
31006    BCS  31100      ; Routine (ASCII-Code größer 58)
31008    JMP     125     ; Zurück zu CHRGET-Routine
31011    ...             ; Hier beginnt die eigentliche
  .        .             ; „!“-Routine, deren Funktion
  .        .             ; Sie selbst festsetzen können
  .        .
31100    RTS             ; Ende „!“ und CHRGET
```

Beachten Sie, daß der Vergleich mit 58 nach der Abfrage auf das neue Befehlswort „!“ geschehen muß, um – falls das eben gelesene Zeichen nicht „!“ ist das Statusregister korrekt einzustellen. Bei gesetztem Carry-Flag erfolgt der Rücksprung zum dem Programmteil, welcher die CHRGET-Routine aufgerufen hat. Bedenken Sie, daß bei Adresse 121 JMP und nicht etwa JSR steht! Das RTS bei 31100 bewirkt also keinen Rücksprung zu CHRGET, sondern in die Routine, die zuvor CHRGET aufgerufen hat. Bedenken Sie, daß die BEQ- und BCS-Anweisungen in 31002 bzw. 31006 maximal 127 Bytes nach vorne verzweigen können (relative Adressierung).

Welche Funktion der neue Befehl „!" nun eigentlich erfüllen soll, haben wir nicht definiert. Sie können dies ab Adresse 31011 selbst vornehmen. Abgeschlossen wird dieser Programmteil mit RTS, wobei der RTS-Befehl natürlich keineswegs genau ab Adresse 31100 stehen muß, sondern eben am Ende der Routine (unter Berücksichtigung der Sprungweite bei Branch-Befehlen).

Wir wollen uns jetzt noch eine Routine ansehen, die unsere „BASIC-Erweiterung" aktiviert. Dabei gehen wir davon aus, daß der Programmteil ab Adresse 31000 bereits vorhanden ist. Dies kann zum Beispiel von einem BASIC-Programm aus mittels POKE erfolgen. Die Aktivierungs-Routine sieht nun wie folgt aus:

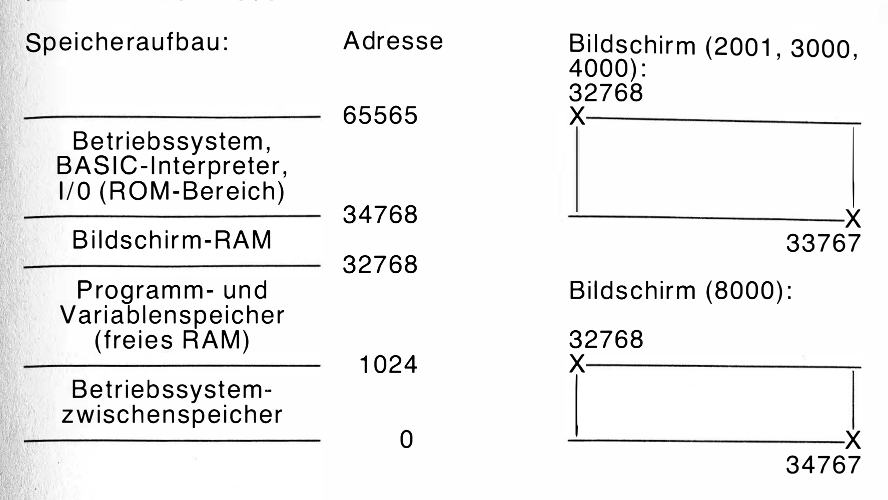
```
690 LDA   # 24     ; Obere BASIC-Grenze in Speicher-
692 STA     52     ; zellen 52/53 (LSB/MSB) mit 31000
694 LDA  # 121     ; festlegen, weil BASIC-Erweiterung
696 STA     53     ; ab Adresse 31000 beginnt.
698 LDA   # 76     ; Assemblercode für JMP nach Adresse
700 STA    121     ; 121 speichern (CHRGET-Routine).
702 LDA   # 24     ; LSB von 31000 als Argument für JMP
704 STA    122     ; nach Adresse 122 speichern.
706 LDA  # 121     ; MSB von 31000 als Argument für JMP
708 STA    123     ; nach Adresse 123 speichern
710 LDA  # 234     ; Assemblercode für NOP nach 124
712 STA    124     ; speichern (nicht unbedingt nötig)
714 RTS            ; Ende der Aktivierung
```

Die angegebenen Adressen beziehen sich wiederum auf den Commodore CBM 8032, für andere Computertypen muß das geändert werden. Der Bereich von 634 bis 825 bildet den sog. „Cassettenpuffer", die nur für Rekorderbetrieb benötigt wird. Ansonsten ist dieser Bereich frei, wir nutzen ihn hier zur Unterbringung unserer Aktivierungsroutine. Zuerst wird die obere BASIC-Grenze festgesetzt, damit unsere „BASIC-Erweiterung" ab Adresse 31000 nicht von BASIC-Text oder Variablen überschrieben werden kann, dann wird die CHRGET-Routine wie besprochen geändert (JMP und NOP einfügen). Nach Aufruf dieser Routine muß von BASIC aus ein CLR gemacht werden, um einige interne Adressen der neuen BASIC-Grenze anzupassen.

Wie bereits gesagt, können neue BASIC-Befehle, die auf diese einfache Weise implementiert werden, nur im Direktmodus und nicht innerhalb eines Programms genutzt werden. Das liegt daran, daß wir nicht unterscheiden, von welcher Stelle des BASIC-Interpreters aus die CHRGET-Routine aufgerufen wird: z.B. Eingabe-Routine, LIST-Routine, Ausführungs-Routine usw. Immerhin haben Sie jetzt bereits den Stand zahlreicher „Toolkits" und anderer „Kits" erreicht, experimentieren Sie selbständig weiter in dieser Richtung!

Professionelle Software, wie zum Beispiel EXBASIC LEVEL II, das ja auch eine BASIC-Erweiterung darstellt, ist natürlich um ein Vielfaches komplizierter und komplexer aufgebaut. Verstehen Sie das nicht als Abschreckung sondern als Ansporn!

Referenz: Commodore CBM 2001/3000/4000/8000



Geschützte Speicherbereiche für Assemblerprogramme:

634 bis 825(1. Cassettenpuffer; frei, wenn kein Rekorder am Cassettenport betrieben wird)
826 bis 1017 (2. Cassettenpuffer; wird aber z.T. von Diskettenbefehlen überschrieben)

Obere BASIC-Grenze: 2001 ; 134/135 (LSB/MSB)
andere: 52/53 (LSB/MSB)

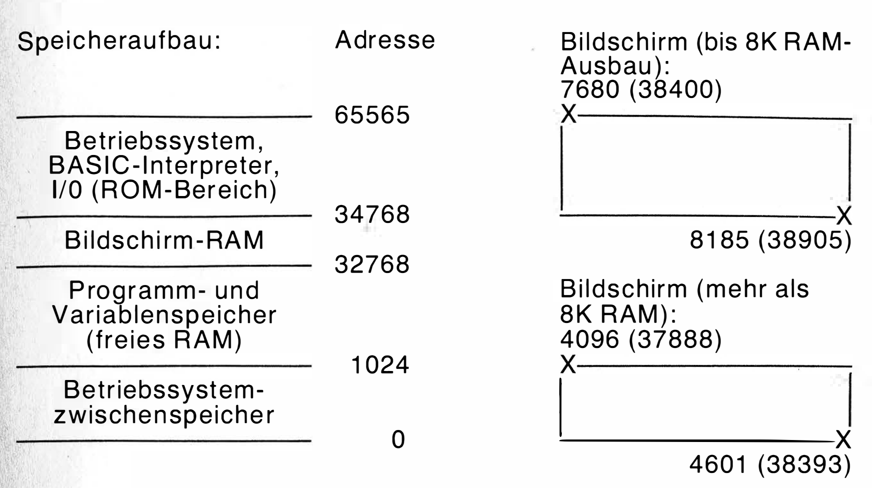
Mit z. B. „POKE 52,24 : POKE 53,121“ wird der Bereich von 31000 bis 32768 von BASIC abgegrenzt und für Assemblerprogramme frei verfügbar (24+256*121=31000; CBM 3000/4000/8000).

CHRGET-Routine (Beginn): 2001 : 194, andere: 112

Interrupt-Vektor (RAM): 2001 : 537/538, andere: 144/145

T.EX.AS. verfügbar seit/ab: ca. 1983

Referenz: Commodore VC-20



Geschützte Speicherbereiche für Assemblerprogramme:

828 bis 1019 (Cassettenpuffer; frei, wenn kein Rekorder am Cassettenport betrieben wird)

1024 bis 4095 (frei, wenn Speicherkapazität größer als 8K, und dennoch ein 3K-Modul eingesteckt ist)

2-Byte-Zeiger in 55/56 (LSB/MSB) legt die obere Grenze des BASIC-Bereiches fest. Mit z.B. „DOKE55, DEEK(55)-1024 wird Bereich von 30001 bis 32767 für Assemblerprogramme frei.

CHRGET-Routine (Beginn): 115

Interrupt-Vektor (RAM): 788/789

T-EX.AS. verfügbar seit/ab: ca. 1983

Referenz: Commodore 64

Speicheraufbau:	Adresse
	65565
Betriebssystem, BASIC-Interpreter, 1/0 (ROM-Bereich	
	40960
Programm- und Variablenspeicher (freies RAM)	2048
Bildschirm-RAM	
	0
Betriebssystem-zwischenspeicher	

Bildschirm:

1024 (55296)
X———————X
2023 (56295)

Die Zahlen in Klammern bezeichnen den Bereich des Farb-RAMs.

Geschützte Speicherbereiche für Assemblerprogramme:

828 bis 1019 (Cassettenpuffer; frei, wenn kein Rekorder am Cassettenport betrieben wird)
49152 bis 53247 (frei, solange nicht Erweiterungsmodule o.ä.diesen Bereich nutzen)

2-Byte-Zeiger in 55/56 (LSB/MSB) legt die obere Grenze des BASIC-Bereiches fest. Mit POKE, DOKE oder HIMEM kann von oben ein Bereich für Assemblerprogramme reserviert werden.

CHRGET-Routine (Beginn): 115

Interrupt-Vektor (RAM): 788/789

T-EX.AS. verfügbar seit/ab: ca. 1983

Referenz: Commodore 600/700

Speicheraufbau:

Bereich	Adresse Bank 15
	65535
Betriebssystem, I/0 (ROM-Bereich)	
	47104
Bildschirm-RAM	
	45056
nicht belegt	
	40960
BASIC-Interpreter (ROM-Bereich)	
	32768
nicht belegt (Cartridge-Bereich)	
	2048
Betriebssystem-zwischenspeicher	
	0

Es existieren mehrere sog. „Banks" mit jeweils 64 K. Bank 15 beinhaltet das Betriebssystem etc. wie nebenstehend aufgeführt, Bank 0 steht vollständig als Speicher für BASIC-Text zur Verfügung, Bank 1 als Speicher für Variablen.

Bildschirm:
(Bank 15)

X (oben links) – X (unten rechts): 45056

Geschützte Speicherbereiche für Assemblerprogramm:

700 bis 760 frei (Stand 9/83, s. Notiz unten)
Obere BASIC-Grenze: 63/64 (LSB/MSB)

CHRGET-Routine (Beginn): (nur im ROM verfügbar)
ROM-Vektor: 47655/47656
(zeigt auf RAM-Vektor)
RAM-Vektor: 656/657 zeigt auf 47666
(Beginn der ROM-Routine)

Interrupt-Vektor (RAM): 768/769

Alle Daten Stand Oktober 1983. Zu diesem Zeitpunkt wurden von Commodore noch Veränderungen am Betriebssystem vorgenommen.

T.EX.AS. verfügbar seit/ab: 1983

Referenz: Apple II/IIe

Speicheraufbau:	Adresse
	65565
Betriebssystem, BASIC-Interpreter, I/0 (ROM-Bereich)	
	49152
Programm- und Variablenspeicher (freies RAM)	
	2048
Bildschirm-RAM	
	1024
Betriebssystem-zwischenspeicher	
	0

Bildschirm (40 Zeichen/ Zeile):
1024
X
X
2023

;„Bildschirm-RAM“ bezeichnet hier die 1. Textseite.
1. Grafikseite: 8192 – 16383
2. Grafikseite: 16384 – 24575

Der Bereich von 38400 bis 49151 wird üblicherweise von DOS (einer Betriebssystemunterstützung) belegt (3 Puffer, 45K).

Geschützte Speicherbereiche für Assemblerprogramme:

768 bis 975 (frei in der Grundversion, wird aber u.U. von Erweiterungsplatinen etc. belegt)

2-Byte-Zeiger in 115/116 (LSB/MSB) legt die obere Grenze des BASIC-Bereiches fest (von BASIC aus mit HIMEM).

CHRGET-Routine (Beginn): 177

Interrupt-Vektor (RAM): 1022/1023
(Der Apple führt keinen regelmäßigen Interrupt durch).

T.EX.AS. verfügbar seit/ab: 1983

Referenz: Atari 400/800

Speicheraufbau:	Adresse
	65565
Betriebssystem, BASIC-Interpreter, I/0 (ROM-Bereich)	
	40960
freier RAM-Bereich wird genutzt für 1. Programm und 2. Variablen, 3. Bildschirm-RAM, 4. DOS 5. Betriebssystem-zwischenspeicher	
	0

Die Aufteilung des RAM-Bereiches bei Atari 400/800 ist nicht zwingend vorgegeben und wird über Vektoren festgelegt. Daher kann die nebenstehende Speicheraufteilung nur wenige absolute Werte beinhalten, die Vektor-Angaben entnehmen Sie bitte Ihrem Atari-Handbuch.

Geschützte Speicherbereiche für Assemblerprogramme:

2-Byte-Zeiger in 725/726 (LSB/MSB) legt die obere Grenze des BASIC-Bereiches fest. Mit POKE, DOKE oder HIMEM kann von oben ein Bereich für Assemblerprogramme reserviert werden.

Interrupt: Der Atari 400/800 führt eine Reihe unterschiedlicher Interrupts aus und kennt daher auch verschiedene Interrupt-Vektoren

T.EX.AS. verfügbar seit/ab: 1984

ADC	Addiert Speicher zu Akkumulator mit Carry
AND	„AND“ Speicher mit Akkumulator
ASL	Linksverschiebung um ein Bit (Speicher, Akkumulator)
BCC	Verzweigt bei gelöschtem Carry-Flag
BCS	Verzweigt bei gesetztem Carry-Flag
BEQ	Verzweigt bei gesetztem Z-Flag (Null-Ergebnis)
BIT	Bit-Test Speicher/Akkumulator
BMI	Verzweigt bei gesetztem N-Flag (Negativ-Ergebnis
BNE	Verzweigt bei gelöschtem Z-Flag (ungleich Null)
BPL	Verzweigt bei gelöschtem N-Flag (positives Ergebnis)
BRK	Software-Interrupt-Auslösung
BVC	Verzweigt bei gelöschtem Overflow
BVS	Verzweigt bei gesetztem Overflow
CLC	Löscht Carry-Flag
CLD	Löscht Dezimalbetrieb-Flag
CLI	Löscht Interrupt-sperren-Flag
CLV	Löscht 0verflow-Flag
CMP	Vergleich mit Akkumulator
CPX	Vergleich mit X-Register
CPY	Vergleich mit Y-Register
DEC	Dekrementiert Speicher um eins
DEX	Dekrementiert X-Register um ein
DEY	Dekrementiert Y-Register um eins
EOR	„Exclusive-Or“ Speicher mit Akkumulator
INC	Inkrementiert Speicher um eins
INX	Inkrementiert X-Register um eins
INY	Inkrementiert Y-Register um eins
JMP	Sprung zu Adresse
JSR	Unterprogrammsprung zu Adresse
LDA	Lädt Akkumulator
LDX	Lädt X-Register
LDY	Lädt Y-Register
LSR	Rechtsverschiebung um ein Bit (Speicher, Akkumulator)
NOP	Keine Operation
ORA	„OR“ Speicher mit Akkumulator
PHA	Legt Akkumulator auf Stack
PHP	Legt Prozessorstatus auf Stack
PLA	Holt Akkumulator von Stack
PLP	Holt Prozessorstatus von Stack
ROL	Linksrotation um ein Bit (Speicher, Akkumulator)
ROR	Rechtsrotation um ein Bit (Speicher, Akkumulator)
RTI	Rücksprung aus Interruptsequenz
RTS	Rücksprung aus Unterprogramm
SBC	Subtrahiert Speicher von Akkumulator mit Carry
SEC	Setzt Carry-Flag
SED	Setzt Dezimalbetrieb-Flag
SEI	Setzt Interrupt-sperren-Flag
STA	Speichert Akkumulator in Speicher
STX	Speichert X-Register in Speicher
STY	Speichert Y-Register in Speicher
TAX	Transfer Akkumulator nach X-Register
TAY	Transfer Akkumulator nach Y-Register
TSX	Transfer Stackpointer nach X-Register
TXA	Transfer X-Register nach Akkumulator
TXS	Transfer X-Register nach Stackpointer
TYA	Transfer Y-Register nach Akkumulator

ADC	Add memory to accumulator with carry
AND	„AND“ memory with accumulator
ASL	Arithmetic shift left one bit (memory or accumulator)
BCC	Branch on carry clear
BCS	Branch on carry set
BEQ	Branch on equal (result zero)
BIT	Test bits in memory with accumulator
BMI	Branch on result minus
BNE	Branch on result not equal (zero)
BPL	Branch on result plus
BRK	Force break
BVC	Branch on overflow clear
BVS	Branch on overflow set
CLC	Clear carry flag
CLD	Clear decimal mode
CLI	Clear inerrupt disable flag
CLV	Claer overflow flag
CMP	Compare memory and accumulator
CPX	Compare memory and index X
CPY	Compare memory and index Y
DEC	Decrement memory by one
DEX	Decrement index X by one
DEY	Decrement index Y by one
EOR	„Exclusive-Or“ memory with accumulator
INC	Increment memory by one
INX	Increment index X by one
INY	Increment index Y by one
JMP	Jump to new location
JSR	Jump to subroutine
LDA	Load accumulator with memory
LDX	Load index X with memory
LDY	Load index Y with memory
LSR	Logic shift „right“ one bit (memory or accumulator)
NOP	No operation
ORA	„OR“ memory with accumulator
PHA	Push accumulator on stack
PHP	Push processor status on stack
PLA	Pull accumulator from stack
PLP	Pull processor status from stack
ROL	Rotate one bit left (memory or accumulator)
ROR	Rotate one bit right (memory or accumulator)
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract memory from accumulator with borrow
SEC	Set carry flag
SED	Sec decimal mode
SEI	Set interrupt disable status
STA	Store accumulator in memory
STX	Store index X in memory
STY	Store index Y in memory
TAX	Transfer accumulator to index X
TAY	Transfer accumulator to index Y
TSX	Transfer stack pointer to index X
TXA	Transfer index X to accumulator
TXS	Transfer index X to stack pointer
TYA	Transfer index Y to accumulator

ADRESSIERUNGSARTEN

	akkumulator	unmittelbar	zero page	zero page ,X	zero page ,Y	absolut	absolut ,X	absolut ,Y	implizit	relativ	(indirekt),X	(indirekt),Y	indirekt
ADC		105	101	117		109	125	121			97	113	
AND		41	37	53		45	61	57			33	49	
ASL	10		6	22		14	30						
BCC										144			
BCS										176			
BEQ										240			
BIT			36			44							
BMI										48			
BNE										208			
BPI										16			
BRK									0				
BVC										80			
BVS										112			
CLC									24				
CLD									216				
CLI									88				
CLV									184				
CMP		201	197	213		205	43	217			193	209	
CPX		224	228			236							
CPY		192	196			204							
DEC			198	214		206	222						
DEX									202				
DEY									136				
EOR		73	69	85		77	93	89			65	81	
INC			230	246		238	254						
INX									232				
INY									200				
JMP						76							108
JSR						32							
LDA		169	165	181		173	189	185			161	177	
LDX		162	166		182	174		190					
LDY		160	164	180		172	188						
LSR	74		70	86		78	94						
NOP									234				
ORA		9	5	21		13	29	25			1	17	
PHA									72				
PHP									8				
PLA									104				
PLP									40				
ROL	42		38	54		46	62						
ROR	106		102	118		110	126						
RTI									64				
RTS									96				
SBC		233	229	245		237	253	249			225	241	
SEC									56				
SED									248				
SEI									120				
STA			133	149		141	157	153			129	145	
STX			134		150	142							
STY			132	148		140							
TAX									170				
TAY									168				
TSX									186				
TXA									138				
TXS									154				
TYA									152				

	Operation, symbolische Darstellung	Statusregister N Z C I D V
ADC	A + M + C → A, C	* * * · · *
AND	A ∧ M → A	* * · · · ·
ASL	C ← 76543210 ← 0	* * * · · ·
BCC	Branch on C = 0	· · · · · ·
BCS	Branch on C = 1	· · · · · ·
BEQ	Branch on Z = 1	· · · · · ·
BIT	A M; M7 → N, M6 → V	M7 * · · · M6
BMI	Branch on N = 1	· · · · · ·
BNE	Branch on Z = 0	· · · · · ·
BPL	Branch on N = 0	· · · · · ·
BRK	Interrupt, PC+2 ↓, P ↓	· · · 1 · ·
BVC	Branch on V = 0	· · · · · ·
BVS	Branch on V = 1	· · · · · ·
CLC	0 → C	· · 0 · · ·
CLD	0 → D	· · · · 0 ·
CLI	0 → I	· · · 0 · ·
CLV	0 → V	· · · · · 0
CMP	A – M	* * * · · ·
CPX	X – M	* * * · · ·
CPY	Y – M	* * * · · ·
DEC	M – 1 → M	* * · · · ·
DEX	X – 1 → X	* * · · · ·
DEY	Y – 1 → Y	* * · · · ·
EOR	A ⊻ M → A	* * · · · ·
INC	M + 1 → M	* * · · · ·
INX	X +Y → X	* * · · · ·
INY	Y + 1 → Y	* * · · · ·
JMP	(PC + 1) → PCL, (PC + 2) → PCH	· · · · · ·
JSR	PC+2 ↓, (PC+1)→PCL, (PC+2)→PCH	alle
LDA	M → A	* * · · · ·
LDX	M → X	* * · · · ·
LDY	M → Y	* * · · · ·
LSR	0 → 76543210 → C	0 * * · · ·
NOP	No Operation	· · · · · ·
ORA	A ∨ M → A	* * · · · ·
PHA	A ↓	· · · · · ·
PHP	P ↓	· · · · · ·
PLA	A ↑	* * · · · ·
PLP	P ↑	von Stack
ROL	← 76543210 ← C ←	* * * · · ·
ROR	→ C → 76543210 →	* * * · · ·
RTI	P ↑, PC ↑	von Stack
RTS	PC ↑, PC + 1 → PC	· · · · · ·
SBC	A – M – Ꞇ → A	* * * · · *
SEC	1 → C	· · 1 · · ·
SED	1 → D	· · · · 1 ·
SEI	1 → I	· · · 1 · ·
STA	A → M	· · · · · ·
STX	X → M	· · · · · ·
STY	Y → M	· · · · · ·
TAX	A → X	* * · · · ·
TAY	A → Y	* * · · · ·
TSX	S → X	* * · · · ·
TXA	X → A	* * · · · ·
TXS	X → S	· · · · · ·
TYA	Y → A	* * · · · ·

Zeichenerklärungen

- A Akkumulator
- X X-Reg.
- Y Y-Reg.
- M Memory, Speicher
- P Prozessor-Statusreg.
- S Stackpointer
- PC Programm Counter, Programmzähler
- PCL PC low
- PCH PC high
- 0 Null
- 1 Eins
- → Transfer nach
- ← Transfer nach
- ↑ Transfer vom Stack
- ↓ Transfer auf den Stack
- + Addition
- – Subtraktion
- ∨ logisch ODER
- ∧ logisch UND
- ⊻ logisch EXKLUSIV ODER
- * Veränderung
- · keine Veränderung
- = gleich
- N Negativflag
- Z Zeroflag, Nullflag
- C Carry
- I Interruptflag
- D Dezimalflag
- V Overflow
- Ꞇ NOT (C)

Stichwortverzeichnis

Mit diesem Werk hat nun auch der völlige Anfänger eine gute Möglichkeit, die 6502-Assembler-Sprache auf leicht verständlichem und doch umfassenden Weg zu lernen. In einer einzigartigen Mischung aus Theorie und Praxis werden die Grundlagen heutiger Mikroprozessoren, alle Anweisungen der 6502-ASSEMBLER-Sprache mit zahlreichen Beispielen sowie die entsprechenden Programmiertechniken vermittelt. Der häufige Vergleich mit BASIC ermöglicht insbesondere dem mit einfachen BASIC-Kenntnissen vorbelasteten Leser einen einfachen, raschen und gründlichen Einstieg in die ASSEMBLER-Sprache. Referenzseiten speziell für Computer von Commodore, Apple und Atari runden das Werk ab. Dieses Buch ist für jeden geeignet, der direkten Zugriff zum „Herz“ seines Computers dem Mikroprozessor haben will.

Ein weiterer Titel von Andreas Dripke, der ebenso didaktisch hervorragend aufgebaut ist, wie seine bereits erschienenen Werke.

ISBN 3-88623-018-x